University of Alberta

Class-Based Testing of Object-Oriented Framework Interface Classes

By

Jehad Al Dallal

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Department of Computing Science

Edmonton, Alberta

Fall 2003

Your file Votre référence
ISBN: 0-612-87928-3
Our file Notre référence
ISBN: 0-612-87928-3

University of Alberta

Library Release Form

Name of Author: Jehad Al Dallal

Title of Thesis: Class-Based Testing of Object-Oriented Framework Interface Classes

Degree: Doctor of Philosophy

Year This Degree Granted: 2003

Date: May 23 , 2003

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommended to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled ***Class-Based Testing of Object-Oriented Framework Interface Classes*** submitted by **Jehad Sami Mohamed Saleh Al Dallal** in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Paul Sorenson, Professor
(Supervisor)

James Hoover, Associate Professor

Ken Wong, Assistant Professor

James Miller, Profes...

Daniel Hoffman, Associate Professor
(External Examiner)

Date: May 23, 2003

Abstract

An application framework provides a reusable design and implementation for a family of software systems. Frameworks are introduced to reduce the cost of a product line (i.e., family of products that share common features). Software testing is a time-consuming, costly, and ongoing activity during the application software development process. Generating reusable test cases for the framework applications at the framework development stage and using the test cases to test part of the framework application whenever the framework is used can reduce the application development time and cost considerably.

This thesis focuses on testing framework applications at the class level using reusable test cases. Specifically, it addresses how to generate the class-based test cases at the framework development stage and use them effectively at the application development stage. The thesis introduces a novel technique to automate the construction of the class-based testing model using the method specifications provided in the hooks and introduces a technique called all paths-state that uses the constructed testing model to generate the class-based reusable test cases at the framework development stage.

At the application development stage, the application developers may need the flexibility to ignore or modify part of the specifications used to generate the reusable class-based test cases and add new specifications not covered by the reusable test cases.

The thesis shows how to deal effectively with such modifications so that testing becomes easy and straightforward in application development.

The introduced techniques are evaluated using applications of four frameworks of two different types. Case studies are used to show the applicability of the introduced techniques and the effectiveness of providing the reusable test cases with the frameworks in reducing the testing cost of the framework applications. The evaluation results also establish the relation between the reusability of the test cases and the type of the framework.

Finally, as part of this thesis research, a prototype tool was developed to support the generation of the class-based reusable test cases at the framework development stage. The tool also deploys, executes, and evaluates the test cases at the application development stage.

Acknowledgement

First and foremost I am thankful to Allah the Almighty (God) for his grace and mercy.

I would like to express my sincere gratitude to my supervisor Dr. Paul Sorenson for his patience, support and continuous encouragement throughout the course of this study. Dr. Sorenson devoted his time and effort to make this study a success. I consider it a matter of great privilege and a rare opportunity to work under his supervision.

I wish to extend my thanks to the supervisory committee members and my external examiner for their time and effort in reading this thesis thoroughly and making many helpful comments and suggestions.

Special thanks are to my parents for their continuous prayers. My thanks also go to my wife and to my children, Ahmed and Usama, for their patience, encouragement and support during this study. Finally, I should not forget to thank my best friend Dr. Samir Ead for his continuous helpful advices.

Table of Contents

Chapter 1 Introduction

1.1. Motivation

A popular goal of software engineering is to develop techniques and tools to assist in design and implementation to meet the market requirements. Meeting time-to-market demands for a software product or application is often vital to the success of an organization or project. In the highly competitive software market, customers seem to demand less time for development while simultaneously expecting better products. Object-oriented framework technology has assisted tremendously in meeting these escalating demands by providing a reusable design and implementation for a family of software systems that share common features [Beck+ 94]. Therefore, instead of designing and implementing the applications from scratch, developers can reuse the design and implementation of the suitable frameworks and complete or extend the frameworks to build their particular applications. However, researchers commonly limit framework reusability to only code and design, which forces the application developers to spend considerable time and effort in testing their applications from scratch. In a typical programming project, approximately half of the effort is spent on testing activities (i.e., validation and verification) [Saleh+ 01]. Therefore, extending the framework reusability to test artifacts can potentially reduce the framework application testing time and increase application quality. Providing the frameworks with reusable test cases makes the frameworks more usable for and marketable to application developers.

Software testing is a critical and important stage of the application software development life-cycle that affects the overall software quality. Typically, many designers and programmers cooperate in developing commercial quality application software. Due to the size and complexity of such applications, they are usually susceptible to errors. By testing we cannot prove the absence of errors in the developed software, but we can establish some degree of confidence that the software does what is supposed to do and does not do what is not supposed to do. Although testing is a costly

and time consuming activity, it has to be performed to produce highly reliable software applications. Application developers can reduce the software testing cost by reusing test artifacts instead of having to develop new test suites from scratch. Reducing the software testing cost while maintaining the same or better software quality reduces the software development time and maintains or enhances the software quality.

The basic element in testing is a test case, which typically consists of a set of inputs, execution conditions, and expected results. Building reusable test cases for the framework applications at the framework development stage increases the framework development time and cost. However, there exists a high probability that the original investment will be recouped after producing a few framework applications. We believe that this investment cannot be fully realized unless the reusable test cases are effective and easy to use in testing the applications. The cost of reusing the test cases must be much lower than the cost of building the test cases from scratch; otherwise, applications developers will prefer to build their own test cases.

The reusability of test cases is shown to be useful in reducing the testing cost in several testing areas including regression testing (e.g., [Harrold+ 01, Hsia+ 97, Kung+ 94a, Kung+ 94b, Kung+ 96, Rothermel+ 94, Rothermel+ 00, White+ 97]), testing subclasses (e.g., [Harrold+ 92, McDonald+ 96, Murray+ 97, Wilkin+ 02]), testing the use of class libraries (e.g., [Binder 99]), testing software product-lines [McGregor 00, McGregor 01]), and testing object-oriented framework applications [Wang+ 00]. In regression testing, the test cases applied to test the original version of the software are reused to test the modified version. Typically, developers do not consider the effect of the possible modifications of the software when they build the test cases for the original version of the software. Therefore, the reusability of the test cases in regression testing depends greatly on the amount and type of the modifications applied on the software. In subclass testing, the superclass test cases are reused to gain the confidence that the inherited superclass features work properly in the context of the subclass. Therefore, the reusable test cases are limited to testing the inherited features and new test cases have to be created from scratch to test the subclass features. The test cases used in testing the

class libraries are supposed to be applied as-is at the cluster testing level. The implementation of the class library for which the reusable test cases are built does not change from one application to another. This application independence is not present for the framework application, which can differ from one application to another. Guidelines for building and using reusable test cases for a software product-line are proposed in [McGregor 00] and [McGregor 01]. However, due to the generality of the product-line testing problem, it is difficult to introduce specific techniques for generating and using reusable test cases for all types of software product-line. Finally, so far, the work proposed in testing the framework applications using reusable test cases (i.e., [Wang+ 00]) is limited to testing the inherited features of the framework classes in the context of the application classes. This thesis extends the reusability of the test cases generated at the framework development stage to test the features of the framework application classes as well as the inherited ones.

1.2. Thesis Framework

To test an object-oriented application, four main testing levels have to be exercised including method testing, class testing, cluster testing, and system testing [Chen+ 01]. At the method testing level, the method responsibilities are considered. At the class testing level, the intraclass interactions and superclass/subclass interactions are examined. At the cluster testing level, the collaborations and interactions between the system classes are exercised. Finally, at the system level testing, the complete integrated system is exercised usually based on acceptance testing requirements. Figure 1.1 depicts the total testing space, showing how the frameworks and the applications developed using the frameworks have to be tested at the four testing levels. This thesis does not address the framework testing at all. Although some work has been completed on framework testing techniques, such as [Binder 00] and [Al Dallal+ 02], much more work is required in this area. Testing object-oriented applications at the method and system levels is similar to conventional program testing [Chen+ 01]. Addressing the cluster testing of the framework applications is considered as future work. This thesis focuses on testing the framework applications at the class testing level only.

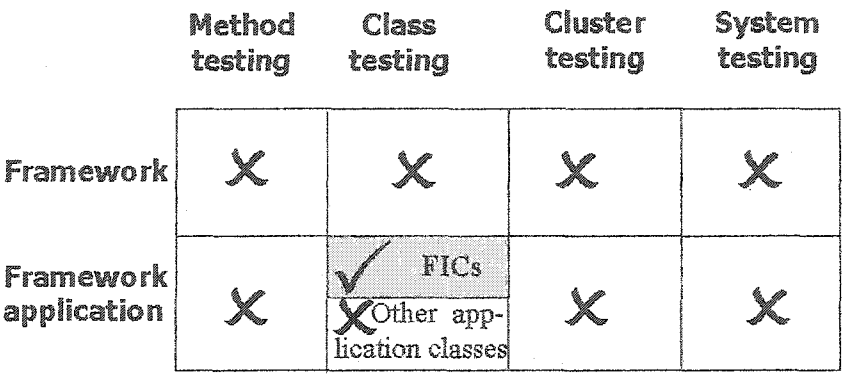

Figure 1.1: The thesis framework

We have identified two types of framework application classes built at the application development stage by application developers: (1) classes that use the framework classes and (2) classes that do not. We call the classes that use the framework classes Framework Interface Classes (FICs) because they act as interfaces between the framework classes and the second type of the classes created by application developers. Instances of FICs are called framework interface objects. FICs use the framework classes in two ways: either by subclassing them or by using them without inheritance. Froehlich [Froehlich 02 and Froehlich+ 97] developed the concept of hooks to show how the framework can be used. Because they define how to use the framework, hooks must also define the FICs and specify the preconditions and postconditions of the FIC methods. Froehlich [Froehlich 02] provided a special purpose language and grammar in which hook descriptions can be written. A hook description includes the implementation steps and the specifications (i.e., preconditions and postconditions) of the FIC methods. Hook points are the places at which the framework users (i.e., application developers) can add their own FICs using the hooks. Figure 1.2 shows the relation between the framework classes, the hook points, the FICs, and the other application classes. In this thesis, we require the framework to be hook-documented (i.e., a complete set of hook descriptions for each FIC is provided). Otherwise, the missed specifications of the FIC methods have to be provided using other framework specification documents or by communicating with the framework developers.

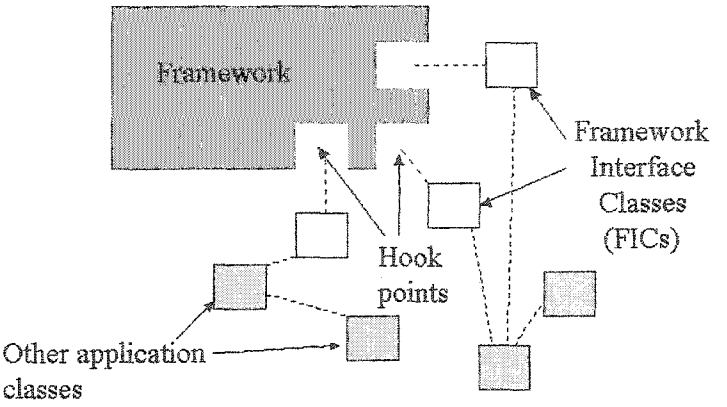

Figure 1.2: Framework application classes

FICs are the only classes built at the application development stage using specifications known at the framework development stage. At the framework development stage, neither the specifications nor the implementations of the other application classes are known. The goal of this thesis is to develop effective techniques and tools for testing the FICs at the application development stage using reusable class-based test cases built at the framework development stage. We focus our testing on the FICs because they are the classes that are given specifications at the framework development stage. Other application classes have to be tested from scratch using the class-based testing techniques (e.g., [Doong+ 94, Hoffman+ 97, Chen+ 98, Binder 99, Ball+ 00, Cheon+ 02, Daley+ 02]).

Testing the FICs achieves two main goals:

1. Increasing the confidence that the implemented methods of the FICs interact properly as described in the hook descriptions.
2. Increasing the confidence that the inherited features of the framework classes work properly in the context of the FICs that extend the framework classes.

FICs are classes built at the application development stage. The implementations of the FICs do not exist at the framework development stage, therefore, we cannot test these classes at that stage. Different implementations of a FIC in different framework applications are developed using the same framework hooks. Therefore, the different

implementations share common specifications provided at the framework development stage. As a result, reusable test cases generated using the common specifications can be built at the framework development stage and used as-is or customized at the application development stage to test the implementations of the FICs.

1.3. Generating Reusable Class-Based Test Cases

In this thesis, we show how to generate reusable class-based test cases at the framework development stage. The test cases are generated using the specifications (i.e., preconditions and postconditions) of the FIC methods provided in the hooks. In specification-based testing, testers use specification testing models such as extended state machines (e.g., [Hoffman+ 97, Binder 99]) or formal specifications (e.g., [Doong+ 94, Chen+ 01]) to generate the test cases. Therefore, two problems have to be considered when generating class-based test cases for the FICs: (1) how to build class-based testing models for the FICs and (2) how to use the models to generate effective test cases.

Unfortunately, the testing models for the FICs are not available at the framework development stage, however, the specifications of the FIC methods are available. Method specifications are typically used as testing oracles to evaluate the results of the test cases (e.g., [Briand+ 02b, Boyapali+ 02, Meyer 92, Cheon+ 02, Jcontract, iContract]). In this thesis, we have extended the use of the method specification to synthesize the state class-based testing models for the FICs. This reduces the chance of errors and cost of generating the class-based test cases and provides a consistent state-based testing model with respect to the specifications of the class methods. We make use of Binder's work [Binder 99] that shows how to express the class behavior using a testable state-based model and we show how to use the specifications of the class methods in building the required testable state-based model. Using the method specifications provided in the hooks for testing purposes adds a new value for the hooks, which were initially introduced for prescriptive documentation purposes. In this thesis, we are using the information provided in the hook descriptions to synthesize the testing models and to evaluate the results of the test cases.

The second problem in generating the test cases is how to use the synthesized testing models in generating the test cases. There are several state-based test case generation techniques introduced in the literature to achieve certain testing coverage for the specification testing model of a class under test (e.g., [Chow 78, Offut+ 99, Binder 99, Bogdanov+ 01, Abdurazik+ 00a]). However, based on specific requirements of an application, an application developer can choose not to implement all the specifications of the FICs, which can then affect the coverage of the already generated test cases using the testing techniques introduced in the literature. This thesis introduces a novel technique called all paths-state that extends round-trip path coverage [Binder 99] to generate test cases from the synthesized class-based testing models of the FICs. The technique solves the problem caused by not fully implemented FIC specifications and therefore, increases the specification coverage of the test cases at the application development stage. We have conducted an empirical evaluation that shows that if the application developer ignores some FIC specifications introduced at the framework development stage, the reusable test cases built using the all paths-state technique always cover the reused specifications, which is not the case for the test cases built using other known state-based testing techniques.

It is important to note that the effort required to build the testing models of the FICs and generate the reusable test cases is expended once at the framework development stage. Each time an application is developed using the framework, this one-time effort spent at the framework development stage is recouped.

1.4. Using the Reusable Test Cases

When developing an application, the application developers need the flexibility to ignore or modify part of the FIC specifications used to generate the reusable class-based test cases and add new specifications not covered by the reusable test cases. In this thesis, we have proposed easy and straightforward ways to use the reusable test cases considering the flexibility that the developer has in modifying the FIC specifications. We have identified the following five problems and proposed effective solutions for each of them:

(1) How to find and discard the test cases for the ignored specifications.

(2) How to map the names of the implemented FIC methods to the names of the FIC methods introduced in the hooks and used in the reusable test cases.

(3) How to test different implementations of the same FIC method introduced in the hooks.

(4) How to deal with the flexibility that the user has in modifying the parameters of the FIC methods introduced in the hooks.

(5) How to test the new specifications added by the application developer.

In addition, the thesis studies the fault coverage of the test cases applied at the application development stage to test the FICs in comparison to the fault coverage of the round-trip path test cases. The thesis also examines experimentally the specification coverage of the reusable test cases and the relation between the percentage of the number of FICs covered by the reusable test cases and the type of the framework which the application uses. The case studies confirm that it is more worthwhile to provide reusable class-based test cases for domain-oriented frameworks compared to application-oriented frameworks.

Finally, we have designed and developed a prototype of a tool to support the generation and the use of the reusable class-based test cases for the FICs. The tool uses the Jcontract tool [Jcontract] to evaluate the test cases using the method specifications.

Figure 1.3 summarizes the FIC testing process as proposed in the thesis. At the framework development stage, the specifications of the FIC methods are used to synthesize the FIC class-based testing model. The model is used to generate the reusable class-based test cases for the FIC. At the framework application development stage, the test cases are used to test the implemented FICs. Finally, the specifications of the FIC methods are used to evaluate the results of the test cases. In the tool prototype, we have implemented the last three steps in the FIC testing process and left the implementation of the first step for future work.

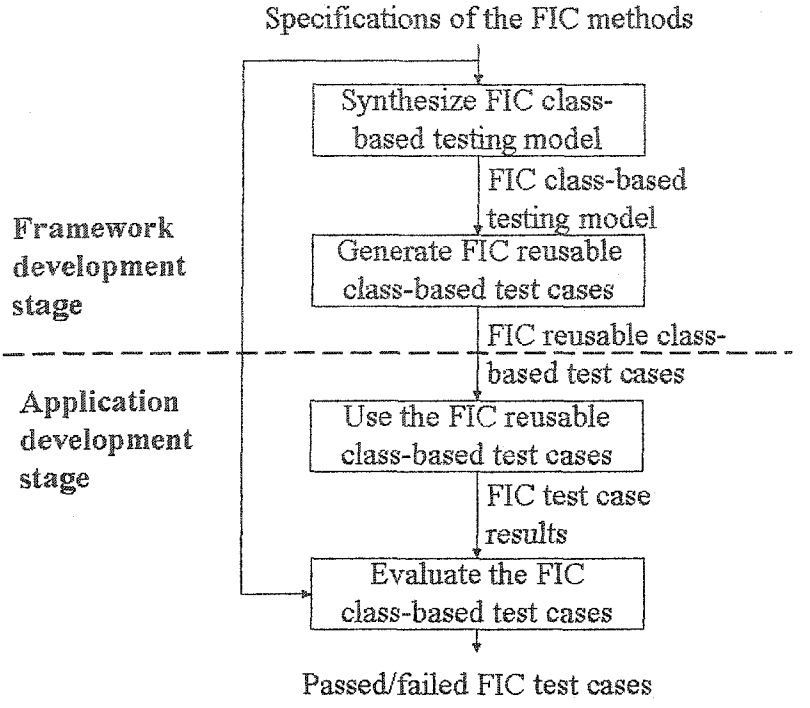

Figure 1.3: The FIC testing process.

1.5. Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 gives an overview of the related research in object-oriented framework technology and software testing. Based on the background literature discussed in Chapter 2, we introduce effective techniques for the generation and use of FIC reusable class-based test cases in chapters 3 and 4. Specifically, Chapter 3 focuses on generating reusable class-based test cases for the FICs at the framework development stage. First, we fill the gap between the inputs of the testing process (i.e., the specifications of the FIC methods provided in the hooks) and the generation process of the test cases by introducing algorithms to synthesize the FIC state-based testing model using the inputs of the testing process. Second, we introduce the all paths-state technique to generate the reusable test cases from the synthesized testing model of the FIC. Chapter 4 proposes easy and straightforward ways to use the reusable test cases generated using the techniques introduced in Chapter 3. In Chapter 5, we present experimental studies to show the specification coverage of the techniques introduced in Chapters 3 and 4. Chapter 6 introduces a tool that implements most of the

theoretical algorithms developed in chapters 3 and 4. Finally, Chapter 7 discusses the thesis contributions and future work.

Chapter 2 Background

There are two main research directions related to the thesis area. The first one is object-oriented frameworks, which includes types of frameworks and framework documentation. The second area of research is software testing, which includes testing object-oriented frameworks and their applications. Not all background material included in this chapter is closely related to the thesis area; however, this chapter gives the reader knowledge of the related work in the broad area of the thesis. Therefore, the included material is useful not only for the particular areas of the thesis, but is helpful in understanding the areas related to some future work described in Section 7.2.

2.1. Object-Oriented Frameworks

An object-oriented framework is the reusable design and implementation of a system or subsystem [Beck+ 94]. It contains a collection of reusable concrete and abstract classes. The framework design provides the context in which the classes are used. The framework itself is not complete. Users of the framework are supposed to complete or extend the framework to build their particular applications. Places at which users can add their own components are called hook points [Froehlich+ 98]. Typically, hooks are associated with *problem domain classes*. Problem domain classes [Binder 99] are representations of external entities or concepts that are necessary for implementation-independent models of the system. For example, in an order-processing system, *Customer*, *Order*, and *Product* are problem domain classes; *LinkedList* is not. When the framework is used to build an application, hooks are used to build classes that extend or use the problem domain classes. We call these classes *framework interface classes* (FICs). Problem domain classes are associated with the hot spot areas of the framework, while the non-problem domain classes are in the frozen spot areas of the framework.

2.1.1. Framework types

Frameworks are classified according to their scope into three categories [Fayad+ 97]: *enterprise application frameworks*, *system infrastructure frameworks*, and *middleware integration frameworks*. The enterprise application frameworks are also known as domain

frameworks and they address different types of applications in a broad application domain such as telecommunications, avionics, manufacturing, and financial engineering. The system infrastructure frameworks are also known as application frameworks and they address different types of applications in different application domains. Moreover, they simplify the development of portable and efficient system infrastructure including frameworks for user interfaces, communication frameworks, and operating systems. Finally, the middleware integration frameworks are also known as support frameworks and they are used to integrate distributed applications and components. ORB frameworks, message-oriented middleware, and transactional databases are common examples of this type of framework. This thesis studies the relationship between the reusability of the FICs and the type of the framework used in building the application under test.

Frameworks are classified according to their customization method into two categories [Johnson+ 88]: white box and black box. In white box frameworks, the functionality is extended or customized by subclassing some existing framework classes. In the black box frameworks, compositions and existing components are used without inheritance. Gray-box frameworks contain the characteristics of both black and white-box frameworks.

2.1.2. Framework documentation

According to Johnson [Johnson 92], there are three types of documentation needed for any framework: documents that describe the purpose of the framework, the use of the framework, and the design of the framework. Campbell and Islam [Campbell+ 92] have worked on documenting the framework design. Design patterns [Gamma+ 95], and exemplars [Gangopadhyay+ 95] are used also to describe the design of the framework. Three papers dealing with the first two documentation types include cookbooks [Krasner+ 88], Johnson's patterns [Johnson 92], and motifs [Lajoie+ 94]. Froehlich et al. [Froehlich+ 97] reported that cookbook solutions are narrative and not structured. The pattern approach documents the purpose and the use of a framework as well as elements of the design. The motifs technique combines the idea of design patterns with Johnson's patterns to provide a more complete description of a framework.

In [Froehlich+ 97] and [Froehlich 02], the issue of documenting the purpose of a framework and how it is intended to be used is described. This is done by using the concept of hooks, which describe how to extend or customize parts of the framework to build an application. Hooks deal with the last two documentation types (purpose and use) in a structured and uniform way and provide an augmented view to design documentation.

Froehlich [Froehlich 02] identified four levels of support provided for the adaptation within the framework: option, template, open, and evolutionary. At the option level, a number of pre-built components are provided within the framework and the developer chooses one without requiring extensive knowledge about the framework. At the template level, the developer supplies parameters to components and follows a well-supported pattern of behavior. At the open level, the developer adds new properties to classes or new classes to the framework or extends the framework functionality. At the evolutionary level, the developer changes parts of the framework code or breaks invariants defined on the framework. Using hook descriptions forces the framework designers to articulate clearly the interfaces to their framework.

Froehlich [Froehlich 02] provided a special purpose language and grammar in which the hook description can be written. Each hook description consists of the following parts.

(1) A unique name.
(2) The requirement (i.e., the problem the hook is intended to help solve).
(3) The hook type.
(4) The other hooks required to use this hook.
(5) The components that participate in this hook.
(6) The preconditions (i.e., the constraints that must be true before the hook can be used or the corresponding code is executed).
(7) The changes that should be made to develop the application.

(8) The postconditions (i.e., constraints that must be true after the hook has been used or the corresponding code is executed).

(9) A general comment section.

It is not necessary to have all the above parts for each hook.

Figure 2.1 shows a hook description example for the creation of an account in a banking framework. The *Initialize Account* hook creates a constructor method for the *NewAccount* class. In the constructor method, the account currency is selected. There are three pre-built classes in the framework for money: *USMoney*, *EURMoney*, and *Money*. Moreover, the user has to specify the bank branches in the system. Finally, the user has to specify the *maxPeriod* variable value.

Name: Initialize Account
Requirement: Initialize an account (i.e., set the currency and bank branches).
Type: Template
Uses: None
Participants: Account(framework), NewAccount(app), Amoney(app);
Preconditions: amount>=0;
Changes:
NewAccount.NewAccount(int amount) extends Account.Account(int amount);
Choose AM from (Money, USMoney, EURMoney);
Create Object Amoney as AM() in MyAccount. NewAccount(int);
Create Object branches as Branches() in NewAccount.NewAccount(int);
Repeat as necessary {
Acquire BranchName: string
NewAccount.NewAccount(int) -> branch.addBranch(BranchName);
}
Acquire maxPeriod : integer domains:0-999999;
NewAccount.NewAccount(int) -> NewAccount.setMaxPeriod(maxPeriod);
Postconditions:
Operation NewAccount. NewAccount (int);
NewAccount.balance>=0;
! NewAccount.frozen;
NewAccount.getUpdate()< NewAccount.MaxPeriod
Comments:

Figure 2.1: Description of the *Initialize Account* hook of a banking framework

The introduced hook description supports the framework application test design through the identification of the FICs, their methods, and the preconditions and postconditions of the FIC methods. These preconditions and postconditions are essential

to determine the FIC behaviors and sequential constraints. Moreover, postconditions hold the expected outputs. The preconditions and postconditions of a method are called the method specifications. When a FIC extends a framework class, the inherited methods are either used in the context of the FIC without modifications or extended. For both cases, the hook descriptions show how to use the inherited methods of the framework classes and identify their pre- and postconditions in the context of the FICs. The thesis uses the method specifications to synthesize the class testing models for the FICs. The testing models are used to generate the reusable test cases for the FICs. In addition, the thesis uses the method specifications to evaluate the results of the test cases.

2.2. Software Testing

Software testing is the process of executing a program with the intent of finding errors. Testing is a time consuming and costly ongoing activity during the application software development process. In a typical programming project, approximately half of the time and cost is spent on testing related activities [Saleh+ 01]. Testing cannot prove the absence of errors, but it increases the level of confidence in the developed software. Central to the testing activities is the design of a test suite.

The basic element of a test suite is a test case that describes the input test data, the test preconditions, and the expected output. [IEEE 829] introduces the typical elements of application programming interface level test cases in SAF (Structured Analysis Form). The syntax includes all necessary items to specify a test case: test identifier, description, components under test, set up, input, and expected results. The expected results include the resulting state, returned objects, output arguments, output messages, and exceptions.

2.2.1. Testing object-oriented software

Object-oriented languages reduce some kinds of errors; however, they increase the chance of others and create new fault hazards. In object oriented programming languages, such as C++ and Java, data and functions (i.e., operations on data) are encapsulated within the object itself. Therefore, a simple assignment can lead to variety of actions. Moreover, inheritance, dynamic binding, overriding and overloading are important

features of the object-oriented programming languages. Although these features make object-oriented software appealing, they create serious difficulties in the testing process.

In object-oriented testing, four testing levels are considered: method, class, cluster, and system. At the method-testing level, the responsibilities of the methods within classes are tested. At the class-testing level, intraclass interactions and superclass/subclass interactions are considered [Binder 99]. At the cluster-testing level, the interfaces among the application classes are exercised to test their collaboration. Finally, at the system-testing level, the actual system functionality is compared with the original requirements. If the system is modified, regression testing is required to check that the modifications have not caused unintended effects. Although this thesis focuses on the class testing level, we provide a brief overview of the other testing levels.

- Method testing

To test the methods inside classes, conventional white and black box testing techniques can be applied. In static white box testing [Beizer 90, Roper 94, Myers 79, Binder 99], tools such as control flow graphs and data flow graphs are commonly used. The control flow graph is a graphical representation of program's control structure, using nodes and direct edges. A node in the control flow graph can be a process, a decision, or a junction node. A *process node* represents a sequence of program statements that are uninterrupted by a decision or a junction. A *decision node* is a program point at which the control flow diverges. Finally, a program point at which the control flow merges is called a *junction node*. When states of program data objects (i.e., killed, defined, or used) are attached to the control flow graph edges, the resultant graph is called a *data-flow graph*.

In dynamic white box testing, program instrumentation is the most commonly used technique for programs written in conventional programming languages [Osterweil+ 78, Chen+ 87, Chan+ 87, Price+ 85, Calliss+ 88] and object-oriented programming languages [Chen+ 95, Boujarwah+ 00, Saleh+ 01]. At run time, the results of the inserted probes are tracked to find the errors.

In black box testing, conventional black box testing techniques such as domain testing, equivalence partitioning, and boundary value analysis can be used. A complete reference on black box testing techniques can be found in [Myers 79, Beizer 95, Binder 99].

Binder [Binder 99] shows how the conventional black box testing techniques can be adapted for object-oriented programs and discusses some other object-oriented specific method testing techniques such as *polymorphic message testing*. In *polymorphic message testing*, tests are developed for a client of a polymorphic server. These tests exercise all client bindings to the server.

- Class testing

Research in generating test cases to test an implementation at class level can be divided into two broad approaches: (1) generating test cases from the source code to achieve a given level of statement, branch, or path coverage, and (2) generating test cases from the formal specification of the implementation. Testing techniques that follow the former approach are called implementation-based testing techniques (also sometimes referred to as white-box testing techniques), while testing techniques that follow the latter approach are called specification-based testing techniques (also sometimes referred to as black-box testing techniques). In this thesis, since the specifications of the FICs are provided, we follow the specification-based approach.

The specification of class behavior can be expressed using state-based models such as finite state machines and UML statecharts [Binder 99]. State-based specifications describe software in terms of states and transitions. A state of an object of a class is an abstraction that models a set of instance variable value combinations that share some property of interest. Typically, two special states have to be presented in any object state-model: alpha and omega. The alpha state represents the object before being constructed. The omega state represents the object after being destroyed. A transition is an allowable two-state sequence. Each transition can be associated with (1) an event (i.e., a call for a class method), (2) a set of predicates, and (3) a set of expected actions. To execute a

transition, the object must be in the accepting state of the transition, the event is executed, and the predicates evaluate to true. The UML syntax for a transition is:

event-name argument-list [guard predicate]/action-expression

There are several state-based specification coverage techniques proposed in the literature including:

1. *All-transitions coverage.* In all-transitions coverage, each transition is covered at least once in some test case. Therefore, to test a transition, the test case requires that the object under test be in the accepting state of the transition. The technique does not put any constraints on how to reach the accepting state. Chow [Chow 78] introduced the all-transition coverage technique for finite state machines and Offut et al. [Offut+ 99] adapted the technique for UML statecharts and compared it experimentally with other specification coverage techniques. Bogdanov et al. [Bogdanov+ 01] used the all-transitions coverage technique to derive test sequences in the presence of hierarchical statecharts.

2. *Transition-pair coverage.* In transition-pair coverage, it is required to cover each pair of adjacent transitions at least once in some test case [Offut+ 99,Chow 78,Abdurazik+ 00a]. Therefore, the transition-pair coverage subsumes the all-transitions coverage.

3. *Full predicate coverage.* In full predicate coverage, it is required to cover each clause in each predicate on every transition, if the clause independently affects the value of the predicate [Offut+ 99, Abdurazik+ 00a]. Offut et al. [Offut+ 99] compared experimentally the full predicate coverage and the transition-pair coverage techniques in terms of fault coverage and showed that the full predicate coverage is more effective than the transition-pair coverage in terms of fault coverage. Abdurazik et al. [Abdurazik+ 00a] compared experimentally the transition-pair coverage and the full predicate coverage in terms of cross scoring (i.e., the difficulty of satisfying one criterion in terms of another) and test set size (i.e., number of test cases required to satisfy the criterion). The comparison results

showed that neither transition-pair coverage nor full predicate coverage had high scores when compared to each other, implying that transition-pair tests offer something different from full predicate tests. Moreover, the comparison results showed that the test set size of the transition-pair coverage technique is larger than the test set size of the full predicate coverage technique. This means that applying the transition-pair coverage technique costs more than applying the full predicate coverage technique.

4. *Round-trip path coverage.* In round-trip path coverage, transition sequences that start and end with the same state and simple paths from *alpha* to *omega* state are covered. A simple path includes only an iteration of a loop, if a loop exists in some sequence. The round-trip path coverage guarantees that each transition in the model is covered at least once and, therefore, it subsumes the all-transitions coverage. The round-trip path strategy was proposed originally by Chow [Chow 78] and was denoted as W-method. Binder [Binder 99] adapted the strategy to UML statecharts and called it round-trip path testing. Antoniol et al. [Antoniol+ 02] showed experimentally that the round-trip path testing strategy is reasonably effective at detecting faults. Kim et al. [Kim+ 99] used a technique similar to the round-trip path strategy to derive testing trees for testing control and data flow through states.

In Section 3.3, we discuss the weakness of the above coverage techniques when used to generate reusable test cases for the FICs and we introduce a new coverage technique that overcomes the weakness. In Section 5.4, the new technique is compared experimentally, in terms of the transition coverage, with the all-transitions and round-trip path coverage techniques.

In [Ball+ 00], an approach for automated testing of container classes based on combinatorial algorithms for state generation is introduced. In [Hoffman+ 97], the ClassBench framework is introduced to support a class-testing approach that traverses a testgraph [Hoffman+ 94], which is a graph representing selected states and transitions of the class under test. Daley et al. [Daley+ 02] introduced a class table-driven testing

approach and a supporting Roast framework for the testing of Java classes. In this approach, effective test data for the test cases are generated automatically.

In [Binder 99], the class flow graph is introduced to provide a testable model for the class behavior. The class flow graph combines the class state model with the control flow graphs of the methods and, therefore, is used in integrated black and white box class testing. In [Beydeda+ 01], the class specification implementation graph is introduced to provide a testable model for the class behavior. The class specification implementation graph combines the control flow graph generated on the basis of the method specifications with the control flow graph generated on the basis of the method implementation. The resulting graph is used in generating integrated black and white box class-based test cases. The implementations of the framework application classes are not available at the framework development stage when the reusable test cases are generated. Therefore, the integrated black and white box class models and testing techniques cannot be applied in our research.

- Cluster testing

Binder [Binder 99] discussed several cluster testing approaches such as the *class association test*, *round trip scenario test*, *controlled exception test*, *mode machine test*, *polymorphic server*, and *modal hierarchy*. In the *class association test*, a UML class diagram is used to develop test cases to test the associations among classes. In the *round trip scenario test*, a control flow model is extracted from a UML sequence diagram and minimal branch and loop coverage is applied to develop the test cases. A *Controlled exception test* verifies exception handling. In the *mode machine test*, the state behavior of a cluster is modeled and a state-based test suite is developed. Finally, the *polymorphic server* and *modal hierarchy* approaches are used to test the class hierarchy.

In [Wu+ 03] and [Badri+ 02], different UML diagrams are used at the cluster testing level. To perform the cluster testing, Wu et al. [Wu+ 03] proposed exercising the direct and indirect interactions between components caused by interface invocations or data dependence relationships. To test these interactions, several approaches are proposed.

The approaches use collaboration, sequence, and statechart UML diagrams. Finally, UML-based test adequacy criteria that use UML models are proposed to build the required test cases. Badri et al. [Badri+ 02] extended the collaboration diagram to capture the specification of exclusion among several messages and the expression of iteration. The extended collaboration diagram is described using the Collaboration Diagrams Description Language, which is used in synthesizing the message control flow. The message control flow is translated into a message tree that shows all possible message invocation sequences. Finally, the tree is used to generate test sequences.

- System testing

Binder [Binder 99] proposed two system testing level techniques: *extended use cases* and *covered in CRUD*. The *extended use cases* technique develops test suites to cover application input/output relationships. The *covered in CRUD* technique exercises all basic operations (i.e., create, read, update, and delete). Briand et al, [Briand+ 02a] proposed a technique to produce system-based test cases using three UML diagrams: use case diagram, sequence diagram, and class diagram. In addition, collaboration diagrams [Addurazik+ 00b] and UML statecharts [Hong+ 00] were found to be useful in testing object-oriented software at system level.

[Binder 96a] is a comprehensive survey of research and practitioner work on testing object-oriented software published up to the end of December 1994. The survey includes over 140 publications.

2.2.2. Contracts as testing oracles

In software testing, it is required to develop oracles to evaluate the actual results of the test cases as pass or no pass. Specification-based testing techniques that use formal specifications support the automation of the output checking as well as input generation. In [Doong+ 94], the prototype testing system ASTOOT is introduced to automatically generate test drivers for Eiffel classes and evaluate the testing results using algebraic specifications. In [Chen+ 98] and [Chen+ 01], integrated black and white class-based testing approaches are introduced. The black-box technique is used to select test cases.

The white-box technique is used to decide whether two objects resulting from executing the test cases are observationally equivalent and to select test cases in some situations. The approaches are based on mathematical theorems. Ball et al. [Ball+ 02] reported that the approaches based on formal specifications are hard to apply in practice because the formal specifications for industrial software are usually unavailable.

Recently, testing researchers started to use an automatic error checking mechanism called contracts [Briand+ 02b, Boyapali+ 02, Meyer 92, Cheon+ 02, Jcontract, iContract] as a substitute to hard-coded testing oracles. Contracts are used to specify the specifications (i.e., preconditions and postconditions) of the class methods and the class invariants. Method preconditions are the conditions that must be true before the method is executed. Method postconditions are the conditions that must be true after the method is executed. Class invariants are the conditions that must exist for all methods. Contracts are used at run-time to detect software faults.

In [Briand+ 02b], it is shown that contracts detect a large percentage of failures (roughly 80% of the faults detected using hard-coded oracles). Moreover, it is shown that the percentage of the detected faults depends on the precision of the contracts. Baudry et al, [Baudry+ 01] showed that the quality of the contracts is more important than their quantity. Finally, in [Briand+ 02b], it is found that the effort involved in isolating a fault is reduced eight fold for programs with contracts as compared to programs without contracts.

There are several tools introduced to support specification-based testing and the use of the contracts. Jcontract [Jcontract] and iContract [iContract] are tools used to evaluate test cases generated for Java programs using Design-by-Contract (DbC) contracts. In [Cheon+ 02], Java Modeling Language JML [Leavens+ 99, Leavens+ 01] is integrated with Junit framework [Junit] to test Java methods. JML is also used in the Korat framework [Boyapali+ 02], where the method specifications are used to generate automatically test drivers (i.e., implementations of the test cases) for Java methods and to check the correctness of the outputs. JTest [JTest] is a tool that uses the DbC contracts to

generate automatically test drivers for Java methods and to check the correctness of the outputs. In [Fenkam+ 02], VDM-SL specification is used to generate black-box test drivers and CORBA-supported VDM oracles for CORBA-compliant programming languages.

In this thesis, we extend the use of the specifications (i.e., preconditions and posconditions) of the FIC methods to synthesize the FIC testing model at the framework development stage. The model is used to generate the reusable class-based test cases. At application development stage, we use the method specifications as testing oracles. Finally, the thesis introduces a tool that supports the generation and use of the reusable class-based test cases. The tool uses the Jcontract tool to evaluate the test cases.

2.2.3. Object-oriented framework testing

In this thesis, we assume that the framework has been tested and we focus on testing the FICs at the class level. Sparks et al., [Sparks+ 96] and Codenie et al., [Codenie+ 97] mentioned the difficulty of the framework-testing problem without proposing solutions. Binder [Binder 99] suggested two different approaches for testing frameworks according to the availability of application-specific instantiations. The first approach, called *New Framework Test*, develops test cases for a framework that has few, if any, instantiations. In this approach, four likely types of defects are checked: incomplete or missing behavior or representation, broken association constraints, control defects, and infrastructure code defects. The approach suggests building a demonstration application that provides a minimal implementation of each use case. Test cases have to be developed to test the demo application using *extended use case* test, *class association* test, and transition coverage for state machine (*N+ test strategy*) or branch coverage on all sequence diagrams.

The second approach, called *Popular Framework Test*, develops test cases for an enhanced version of a framework that has many application-specific instantiations. Three classes of defects are more likely in such frameworks: feature interaction defects, compatibility defects, and latent defects not discovered before. Moreover, the new

features can suffer from the same types of defects that occur in any other system (i.e., types of defects checked in New Framework Test approach). To test the new features, the four testing techniques used in testing new frameworks can be used. To test the feature interaction (i.e., the effect of using the old and new features together), an application that develops both old and new features has to be implemented. Finally, to test the feature compatibility (i.e., ensuring that new features do not break old ones) an existing application that implements the old features has to be executed.

Al Dallal et al [Al Dallal+ 02] proposed a technique called Testing Frameworks Through Hooks (TFTH) to generate a test suite to test hook-documented object-oriented frameworks at the system level. The test suite is designed to test the framework implementation at the system level as well as the framework hooks. The technique uses an extended state model to model the FICs and a construction flow graph to model the construction sequence of the hook methods. Round-trip path trees are generated from the state models of the FICs. The trees and the construction flow graphs are traversed to produce the required test suite. The test suite includes test cases that test the framework's open functionalities. Such test cases can be applied at the application testing stage to test whether the implemented open functionalities violate their constraints defined in the hook descriptions. Using the TFTH technique, a large portion of the testing process is automated. The scalability of the TFTH technique is not addressed. In addition, more case studies are required to evaluate the fault coverage of the technique and to compare it with other testing techniques such as the New Framework Test.

2.2.4. Testing framework applications

Binder [Binder 96b] suggested that the testing of framework applications should be based on system requirements. The new classes and objects developed by the application developer must be individually tested, which can be accomplished using the FREE test design methodology [Binder 99]. Moreover, cluster testing should be applied to verify that the developer objects are making correct use of the framework code. In this step, the framework test suite could be extended to test the application extensions.

Binder neither suggested a specific methodology that makes use of the framework test suite to test the applications at the class or cluster level nor provided a discussion on which framework test suite can be extended or how a framework test suite can be extended.

Tsai et al. [Tsai+ 99] discussed the issues of testing applications developed with design patterns using object-oriented frameworks. They have classified the patterns into static and extensible. Static patterns are the ones that do not allow easy extension. The extensible patterns allow the functionality of the application to change. The paper addressed the extensible patterns testing from two viewpoints: the framework developers and the application designers. Framework developers should test that the extensible patterns do allow the application developer to extend its functionality. The application designers should verify that the extension points are properly coded and tested. The paper introduced a technique to generate scenario templates that can be used to generate different types of cluster-based test scenarios. These test scenarios are used to test sequence constraints on the interaction between framework objects and custom objects. Tsia et al's work in testing framework applications is limited to cluster-level testing, which is out of scope of this thesis.

Wang et al. [Wang+ 00] proposes providing the framework with reusable test cases that can be applied at the application development stage. However, these test cases are limited to testing that the inherited framework features work correctly in the context of the application classes that inherit them and it does not address testing the features of the application classes. This thesis extends the use of the reusable test cases to test the features defined in the FICs as well as the inherited features.

2.2.5. Reusability of object-oriented tests cases

There are several testing areas for which reusability of test cases for object-oriented software are proposed and discussed including regression testing, testing subclasses, testing the use of class libraries, testing software product-lines, and testing object-oriented framework applications.

In regression testing, a modified version of the software is tested to provide confidence that the changed parts are behaving as intended and the unchanged parts are not affected by the modifications in an unforeseen way. The test suite used to test the original version of the software or part of it is reused to test the modified version. In attempting to reuse the test suite or part of it, two problems have to be tackled: which test cases of the original test suite can or should be used to test the modified version and which new test cases must be developed to test parts of the modified software [Harrold+ 01]. A number of regression testing techniques have been developed for testing object-oriented applications (e.g., [Harrold+ 01, Hsia+ 97, Kung+ 94a, Kung+ 94b, Kung+ 96, Rothermel+ 94, Rothermel+ 00, White+ 97]). As far as we know, all regression testing techniques are code-based (i.e., based on using the source code analysis to determine the test cases). None of the proposed regression testing techniques determines the test cases that have to be reapplied for the modified classes using the software specifications only. Moreover, researchers in the area of regression testing have focused on finding the original test cases that can or should be used to test the modified version of the software. No specific approaches are proposed for the augmentation of the test cases to test new software parts.

In subclass testing, the superclass test suite or part of it has to be reapplied to gain confidence that the inherited superclass features work correctly in the context of the subclass. In [Harrold+ 92], the knowledge of how the subclass is derived from the superclass is used to determine where the superclass test suite must be changed and which superclass test cases have to be rerun to test the subclass. In [McDonald+ 96], it is shown how the superclass test suite should be changed to test the subclasses and a framework is introduced to execute the test cases and check their results. In [Murray+ 97], the Test Template Framework, a framework for specification-based testing, is extended to include inheritance. Conditions under which superclass test cases can be reapplied as-is or after some modifications are identified. In [Wilkin+ 02], JUnit, a framework for Java unit testing, is extended such that superclass test cases are automatically extracted in subclass drivers. In subclassing, superclass features are

inherited without modifications, redefined, or extended. Therefore, the problem of detecting broken test cases for ignored specifications is not applicable and, therefore, not studied in reusing superclass test cases.

In testing the use of the class libraries and the frameworks, Binder [Binder 99] stated that the class library user and the framework user can reuse the class libraries test suite and the framework test suite, respectively, at cluster testing level without introducing new specific approaches.

In [McGregor 00] and [McGregor 01], the issue of product-line testing is considered. It is suggested to build general reusable test cases, associate them with the software specification, store them in a database at the product line level, and then specialize the test cases for each product. This way of using the test cases is called vertical reuse. In addition, each time a product is constructed and tested, the specialized test cases and the new test cases applied for testing the product are stored in the database. Whenever, the product components are used in building other products, the stored test cases are reused as-is. This way of using the test cases is called horizontal reuse. Due to the generality of the product-line testing problem, it is difficult to introduce specific techniques for generating and using reusable test cases for all types of software product-line.

In [Hornstein+ 02], it is suggested to put built-in-tests, in the form of methods that provide information to test the component, inside the reusable components and to build component testers. The component testers exercise the built-in-tests to test the component. Whenever the components are used, the component testers can be used as-is or modified to verify that the components work correctly in their deployment environment. Three stages in which the component testers can be used are introduced: when the component under test is deployed, during normal execution, and in maintenance. This technique works well for the components used as-is, but it does not work for the components that have their specifications modified at deployment time. In this case, the modifications of the component specifications have to be reflected in the reusable component testers. In [Wang+ 00], built-in tests are incorporated into object-

oriented frameworks and used when the framework classes are inherited at the application development stage. The approach limits the reusability of the built-in tests to verify that the inherited framework features work correctly in the context of the application classes that inherit them. The work proposed in this thesis extends the reusability of the test cases provided with the framework to test the FICs. Testing the FICs includes testing the inherited features from the framework classes in the context of the FICs and testing the new FIC functionalities introduced in the hooks for the FICs. Moreover, the thesis shows how to test the FICs that do not inherit framework classes using reusable test cases.

2.3. What Remains?

Despite the fact that the framework application testing area has received some attention, several related points are either poorly addressed or not addressed. In this thesis, testing framework applications is addressed at the class level using reusable test cases generated at the framework development stage. More specifically, the following key issues are examined:

1. The construction of a class state-based testing model using the method specifications (i.e., preconditions and postconditions) provided in the hook descriptions.
2. The introduction of an effective test case generation technique in terms of fault coverage and transition coverage of the constructed testing model. The generation technique is used at the framework development stage to generate the reusable test cases.
3. The provision of a straightforward and easy way to use the reusable test cases at the framework application development stage.
4. The conducting of case studies to demonstrate the reduction of class testing costs at the framework application testing stage using the reusable test cases.
5. The conducting of case studies to establish a relation between the number of FICs in the framework applications and the type of the framework used.

6. The introduction of a supporting tool for generating the reusable test cases at the framework development stage and deploying, executing, and evaluating the test cases at the application development stage.

Chapter 3 Generating Reusable Class-Based Test Cases

3.1. Introduction

At the class testing level, a testing model has to be constructed and used to generate the test cases. This chapter introduces a novel technique to synthesize a class state-based testing model from the specifications (i.e., preconditions and postconditions) of the FIC methods defined in the hooks. In addition, this chapter introduces a novel test case generation technique called all paths-state to generate test cases that are effective at the application testing stage in covering the reused FIC specifications. The test cases have at least the same fault coverage as the test cases generated using the round-trip path technique. In terms of the overall testing process, this chapter shows how to perform the processes at the framework development stage shown in Figure 1.3.

The chapter is organized as follows. Section 3.2 introduces the class state-based testing model construction technique. The all paths-state testing technique is introduced in Section 3.3. Finally, Section 3.4 provides a summary discussion.

3.2. Automatic Construction of a Class-Based Testing Model

Building a testing model to express the behavior of a class is an essential step for the generation of the class-based test cases. Object-oriented software is well suited to state-based testing. A class behavior can be expressed in a state-transition model, which consists of states and transitions and can be represented using State Transition Diagrams (STD) or UML statecharts. A state-transition model can be easily understood and is widely used in specification-based testing techniques.

As shown in Figure 1.3, the input to the testing process of the FICs is the specifications of the FIC methods. Hook descriptions provide the specifications of the FIC methods in terms of pre- and postconditions. There are two types of pre- and postconditions: (1) construction ones and (2) execution ones and these are illustrated in the example in Figure 3.1. The construction pre- and postconditions are the constraints

that must be satisfied before and after the hook is used, respectively, and they are identified in the hook description by keywords such as *Object*, *Class*, and *Operation*. The execution pre- and postconditions are the dynamic constraints that must be satisfied before and after the methods defined in the hook are executed, respectively. The construction pre- and postconditions, in contrast with the execution ones, do not describe the behavior of the methods defined in the hooks and, therefore, cannot be used in synthesizing the behavioral model of the FIC. The execution preconditions are described in terms of class instance variables and method input parameters. The execution postconditions are described in terms of class instance variables, input parameters, output parameters, method return values, and method thrown exceptions. More precisely, the execution method specifications (i.e., pre- and postconditions) check:

(1) whether class instance variables, method input parameters, method output parameters, and method return values are within the allowed domain of values and

(2) whether the relationships among the values of the class instance variables, method input parameters, method output parameters, and method return values are satisfied.

To help understand the relationship between the hooks and the FICs let us examine a concrete example. Figure 3.1 shows the description of the *Initialize Account* hook of a banking framework. In the *Changes* section of the hook, the FIC called *NewAccount* is introduced. The hook specifies also one of the FIC methods, which is the constructor method *NewAccount(int amount)*. In the *Preconditions* and *Postconditions* sections of the hook description, the preconditions and postconditions of the constructor method are specified. The first stated postcondition is a construction postcondition because it describes a condition that must be satisfied when the hook is used and it is identified by the keyword *Operation*. The other pre- and postconditions are execution ones because they describe the conditions that must be satisfied, respectively, before and after the constructor method is executed. To determine the behavior of a FIC we have to consider all the framework hooks that specify the FIC methods. Appendix D provides the hook

descriptions that specify the methods of the *NewAccount* FIC. For the rest of this thesis, unless stated otherwise, all pre- and postconditions referred to are of type execution, because these are the conditions in which we are most interested in building test cases.

Name: Initialize Account
Requirement: Initialize an account (i.e., set the currency and bank branches).
...
Preconditions: amount>=0;
Changes:
NewAccount.NewAccount(int amount) extends Account.Account(int amount);
...
Postconditions:
1. Operation NewAccount. NewAccount (int);
2. NewAccount.balance>=0;
3. ! NewAccount.frozen;
4. NewAccount.getUpdate()< NewAccount.MaxPeriod

...

Figure 3.1: The description of the *Initialize Account* hook of a banking framework

In our concrete example, the hooks of the banking framework define several public methods for the *NewAccount* FIC. The methods are as follows.

(1) *NewAccount*: a construction method.
(2) *balance*: to inquire about the balance of the *NewAccount*.
(3) *deposit*: to deposit money to the *NewAccount*.
(4) *withdraw*: to withdraw money from the *NewAccount*.
(5) *freeze*: to freeze the *NewAccount*.
(6) *unfreeze*: to unfreeze a frozen *NewAccount*.
(7) *activate*: to activate an inactive *NewAccount*.

The pre- and postconditions of the *NewAccount* FIC method defined in the hooks are listed in Table 3.1. From the preconditions and postconditions, we synthesize the states and transitions of the *NewAccount* class.

Method	Preconditions	Postconditions
NewAccount(amount)	amount>=0	NewAccount.balance>=0 && ! NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod
balance()		
deposit(amount)	! NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod	NewAccount.balance=amount+ NewAccount.balance && ! NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod
withdraw(amount)	NewAccount.balance>=0 && ! NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod	NewAccount.balance= NewAccount.balance-amount && ! NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod
freeze()	! NewAccount.frozen && NewAccount.balance>=0	NewAccount.frozen && NewAccount.balance>=0
unfreeze()	NewAccount.frozen && NewAccount.balance>=0	! NewAccount.frozen && NewAccount.balance>=0 && NewAccount.getUpdate()< NewAccount.MaxPeriod
activate()	NewAccount.balance>=0 && !NewAccount.frozen &&NewAccount.getUpdate()>= NewAccount.MaxPeriod	NewAccount.balance>=0 && !NewAccount.frozen && NewAccount.getUpdate()< NewAccount.MaxPeriod

Table 3.1: The preconditions and postconditions of the *NewAccount* class methods.

Some class methods may not have preconditions, which means that they can be called at any time during the object life cycle. There are three types of variables used in the preconditions and postconditions: (1) non-static instance variable (i.e. an instance variable whose value can change during the object life cycle), (2) static instance variable (i.e. an instance variable whose value cannot be changed during the object life cycle), and (3) local variable (i.e., a variable defined in the parameter list of the method). The return of a method is considered as a non-static instance variable.

For example, in Table 3.1, *balance()* is a method that has no preconditions which means that it can be called at any time during the object life cycle. *MaxPeriod* is a static instance variable, *amount* is a local variable, and the variables *balance*, *frozen*, the return of the *balance()* method, and the return of the *getUpdate()* method are non-static instance variables. The *balance()* method returns the value of the *balance* instance variable. The

getUpdate method calculates the difference between the current date and the last activity date.

Despite the fact that the method specifications hold the specifications for the class behaviors, researchers seem to limit the method specification use to support the automated detection of software failures and the isolation of faults, and to generate method-based test cases. We are not aware of any work that uses the method specifications to generate class-based test cases. For example, in [Cheon+ 02], to test a class behavior, class behavior testing models (e.g., state-transition model or UML statechart) used to generate the test drivers have to be pre-provided, and the method specifications are used only as testing oracles. Hand-construction of the class behavior testing model is expensive, error-prone, and may result in constructing an inconsistent model with the specifications of the class methods, which misleads verification results.

In this thesis, a new technique is introduced to automatically synthesize the state-transition testing model of the FIC sequential class behavior from the specifications of the class methods. This reduces considerably the class testing cost and the chance of errors. The result is a state-based testing model that is consistent with respect to the specifications of the class methods. Therefore, using the introduced state-transition model synthesis technique, only the specifications of the FIC methods have to be provided to test the FIC behavior. The state-transition model is synthesized automatically from the method specifications. After that, a specification-based testing technique can be applied to derive the test drivers (i.e., implementations of the test cases) from the synthesized state-transition model. Finally, the test drivers are executed and the method specifications are used as testing oracles to evaluate the actual results of the test cases as pass or no pass. Figure 3.2 compares the testing process that uses our proposed modeling technique and the one that does not (e.g., [Binder 99, Offut+99, Abdurazik+ 00a]). In process (a), the FIC testing model as well as the FIC method specifications have to be provided to test the FICs, while in process (b), only the FIC method specifications have to be provided to test the FICs and the FIC testing model is synthesized internally in the process.

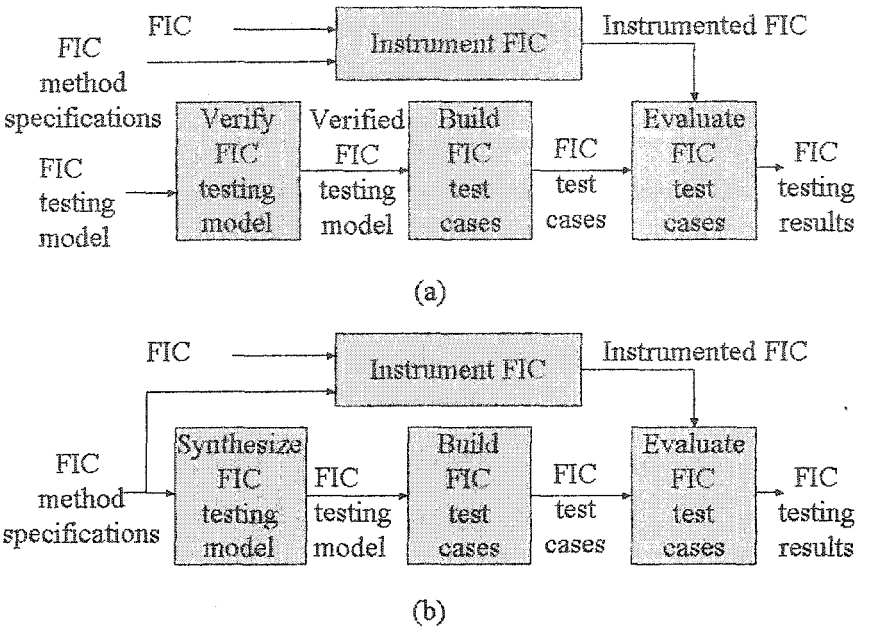

Figure 3.2: FIC behavior testing process using method specifications: (a) without using the proposed modeling technique and (b) with using the proposed modeling technique

3.2.1. Synthesizing the states of a FIC

To synthesize the states of a FIC, we have to construct the condition/instance-variable table. In the table, the columns and rows represent the non-static instance-variables of the FIC and the precondition/postconditions of the FIC methods that contain conditions involving the non-static instance variables, respectively. In the table, we consider only the non-static instance variables. The values of the static instance variables do not change during the object life cycle and, therefore, they do not contribute in determining the object states. Table 3.2 shows the condition/instance-variable table extracted from Table 3.1.

Finally, the table has to be optimized by eliminating redundant rows and unnecessary information. To optimize the table follow these steps

Step 1: Delete any clause that depends on a dynamic variable (i.e., a variable that its value is not assigned at compilation time or can change during the object

life-cycle) because the combinations of the instance variable values that represent a state of a class are determined at compilation time and do not change during the object life-cycle. The deleted clauses are considered later in the transition synthesis process.

Step 2: Delete redundant rows.

Step 3: If the combinations of instance variable values in a row r_1 overlap with the combinations of instance variable values of another row r_2, replace r_1 and r_2 with three rows: the first one contains the combinations of instance variable values contained in r_1 and not contained in r_2, the second one contains the combinations of instance variable values contained in r_2 and not contained in r_1, and the third row contains the overlapping combinations of instance variable values.

Step 4: Iterate through steps 2 and 3 until no redundant rows and no rows for which Step 3 can be applied exist.

Condition identifier	Source of the condition	Non-static instance-variables		
		frozen	balance	getUpdate()
1	Postcondition of the *NewAccount* method, precondition of the *withdraw* method, postcondition of the *unfreeze* method, and postcondition of the *activate* method	false	>=0	<MaxPeriod
2	Precondition of the *deposit* method	false		<MaxPeriod
3	Postcondition of the *deposit* method	false	balance+amount	<MaxPeriod
4	Postcondition of the *withdraw* method	false	balance-amount	<MaxPeriod
5	Precondition of the *freeze* method	false	>=0	
6	Postcondition of the *freeze* method and precondition of the *freeze* method	true	>=0	
7	Precondition of the *activate* method	false	>=0	>=MaxPeriod

Table 3.2: The condition/instance-variable table extracted from Table 3.1

Each row in the optimized table represents a state of the object of that class during its life-cycle based on the instance variable value combination shown in the row. Step 1 ensures that the states do not change during the object life-cycle (i.e., a basic property of the states of a class state-based model as described in [Binder 99]) and, therefore, each remaining clause in the table has boundary values determined at compilation time. The second step ensures that there are no redundant states in the synthesized model. Step 3 ensures that each synthesized state is exclusive (i.e., there is no overlapping states).

Determining the overlapping combinations of instance variable values contained in two rows is straightforward because the condition clauses in the rows have fixed boundary values as ensured in Step 1. Finally, Step 4 holds the stopping criterion for the state synthesis process.

When the optimization rules are applied on Table 3.2, according to Step 1, 'balance+amount' and 'balance-amount' are deleted from rows 3 and 4, respectively, because they depend on dynamic variables. This makes rows 3 and 4 redundant with row 2 and, therefore, rows 3 and 4 are deleted according to Step 2. The combinations of instance variable values contained in row 1 overlap with the combinations of instance variable values contained in row 2. Therefore, to avoid the creation of overlapping states, rows 1 and 2 are substituted with three rows. The first row contains the combinations of instance variable values contained in row 1 and not contained in row 2. This row is ignored because it does not contain any combination of instance variable values (i.e., all the combinations of instance variable values contained in row 1 overlap with the combinations of instance variable values contained in row 2). The second row contains the combinations of instance variable values contained in row 2 and not contained in row 1 (i.e., the combinations of instance variable values that has *balance*<0). The third row contains the overlapping combinations of instance variable values of rows 1 and 2 (i.e., the combinations of instance variable values that has balance >=*0*). Finally, the combinations of instance variable values contained in the third row formed in the previous step overlap with the combinations of instance variable values contained in row 5. Therefore, the two rows are replaced with three rows. The first row is ignored because it does not contain any combination of instance variable values (i.e., all of the combinations of instance variable values contained in the third row formed in the previous step overlap with the combinations of instance variable values contained in row 5). The second row has *getUpdate()* >=*MaxPeriod* and the third row has *getUpdate()* <*MaxPeriod*. This results in having the optimized table shown in Table 3.3. In this table, each row represents a state that corresponds to the instance variable value combinations given in the row. The combinations of instance variable values in each row are called the *state-invariants* [Binder 99].

State identifier	Instance-variables		
	frozen	**balance**	**getUpdate()**
1	false	>=0	<MaxPeriod
2	false	<0	<MaxPeriod
3	false	>=0	>=MaxPeriod
4	true	>=0	

Table 3.3: Optimized table constructed by applying optimization rules on Table 3.2.

3.2.2. Synthesizing the transitions of a FIC

To extract the transitions that model the legal behavior of a FIC, we have to map the preconditions and postconditions of the FIC methods to the extracted state-invariants. Each state in which its state-invariants satisfy the preconditions of a method is a source state for the transition associated with the method call. Moreover, each state in which its state-invariants satisfy the postconditions of a method is a destination state for the transition associated with the method call. The procedure shown in Figure 3.3 explains the mapping process and shows how to extract the predicates and actions of the transitions. The source state of the constructor method is by default the *alpha* state. If no destructor method is specified in the class, an unlabeled transition has to be added from each state, other than the *alpha* state, to the *omega* state.

When the procedure shown in Figure 3.3 is applied on tables 3.1 and 3.3, the transitions shown in Table 3.4 are extracted. For example, since the *balance* method has no preconditions and its postcondition satisfies all the invariants of all states, a self-loop transition associated with *balance()* event is added to each state (other than alpha and omega). The postcondition of the *balance* method is not specified in any state and, therefore, it is added as an action to all the self-loop transitions. For the *withdraw* method, the preconditions satisfy the invariants of state 1 and the postconditions satisfy the invariants of states 1 and 2. Therefore, two transitions associated with the *withdraw* event are added as shown in Table 3.4. Note that the set of preconditions of the method is the same as the invariants of state 1, which causes no predicates to be added at this step. The set of postconditions of the *withdraw* method includes "balance=balance-amount" which is not included in the state invariants of states 1 and 2. Therefore, the postcondition is added to both transitions as actions. Finally, since state 1 now has two outgoing

transitions that have the same labels, the difference between the invariants of the destination states of the transitions has to be added as predicates to the transitions. The difference between states 1 and 2 is that state 1 has balance>=0, while state 2 has balance<0. Therefore, "balance (at destination state) >=0" has to be added to one of the transitions (from state 1 to state 1) and "balance (at destination state) <0" has to be added to the other transition (from state 1 to state 2). Since the predicates are checked at the source states, we can substitute "balance (at destination state)" by "balance-amount".

Inputs: Invariants of the FIC states and the preconditions and postconditions of the FIC methods.

Output: Transitions of the FIC state-based model.

Procedure:

1. *for* each FIC method *do*
2. Search for all states whose state-invariants satisfy the preconditions of the method.
3. Search for all states whose state-invariants satisfy the postconditions of the method.
4. Create a transition from each state found in Step 2 to each state found in Step 3 and associate the method name with the transition as an event.
5. *for* each transition created in *Step 4 do*
6. *if* the set of preconditions of the method is a superset of the set of state-invariants of the source state of the transition *then* add the non-overlapped portion of the set as predicates to the transition.
7. *if* the set of postconditions of the method is a superset of the set of state-invariants of the destination state of the transition *then* add the non-overlapped portion of the set as actions to the transition.
8. *if* there is another transition that has the same source state, event, and predicates *then* add to each of the transitions the difference between the postconditions of the method called in the transition and the state-invariants of the destination states as predicates to the transitions.

Figure 3.3: Construction process of the transitions of the FIC specification model.

Since the transition synthesis method uses the method specifications to synthesize the transitions, it is limited to event-driven transitions (i.e., transitions that have associated events). For non-event-driven transitions, it is required to determine the source and destination states first. The state-invariants of the destination state, which are different than the state-invariants of the source state, are then added as predicates to the transition. For the *NewAccount* class example, we have identified two non-event-driven transition examples as shown in Table 3.5. The invariant of state 3, which is different than the invariant of state 1, is "getUpdate()>=MaxPeriod". Therefore, this difference is added as a predicate to the non-event-driven transition that has the states 1 and 3 as source and destination states, respectively. The same situation applies for the transition that has the states 4 and 3 as source and destination states, respectively.

Transition identifier	Source state identifier	Destination state identifier	Transition event	Transition predicates	Transition actions
1	alpha	1	NewAccount	amount>=0	
2	1	1	balance		return balance
3	2	2	balance		return balance
4	3	3	balance		return balance
5	4	4	balance		return balance
6	1	1	deposit	balance+amount>=0	balance=balance+amount
7	1	2	deposit	balance+amount<0	balance=balance+amount
8	2	1	deposit	balance+amount>=0	balance=balance+amount
9	2	2	deposit	balance+amount<0	balance=balance+amount
10	1	1	withdraw	balance-amount>=0	balance=balance-amount
11	1	2	withdraw	balance-amount<0	balance=balance-amount
12	1	4	freeze		
13	3	4	freeze		
14	4	1	unfreeze	balance>=0	
15	3	1	activate		
16	1	Omega			
17	2	Omega			
18	3	Omega			
19	4	Omega			

Table 3.4: Transitions of the *NewAccount* FIC extracted using the procedure shown in Figure 3.3

Transition identifier	Source state identifier	Destination state identifier	Transition predicates	Transition actions
20	1	3	getUpdate()>=MaxPeriod	-
21	4	3	getUpdate()>=MaxPeriod && !frozen	-

Table 3.5: Non-event-driven transitions of the *NewAccount* FIC

Figure 3.4 shows the synthesized state-transition model represented in a State Transition Diagram (STD). In the diagram, states and transitions are represented by nodes and edges, respectively. To make the diagram more understandable, meaningful names can be associated to the states. In Figure 3.4, states numbered 1, 2, 3, and 4 are named *Open*, *Overdrawn*, *Inactive*, and *Frozen*, respectively. The transitions are labeled by their identifiers shown in tables 3.4 and 3.5.

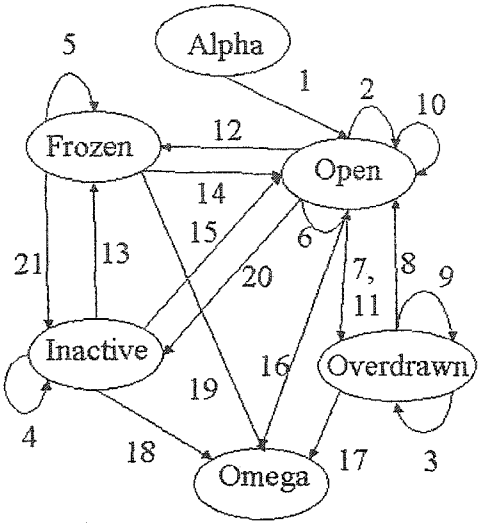

Figure 3.4: STD of the synthesized state-transition model of the *NewAccount* FIC.

The STD given in Figure 3.4 shows the legal behavior of the *NewAccount* FIC. There are several sources for the illegal behaviors of a class including invoking methods at states that do not accept them and invoking methods while setting the environment in a way that causes the methods to throw an exception. The procedure given in Figure 3.5 illustrates the synthesis method of the transitions that model the illegal behaviors of a class. The procedure examines the events and predicates associated with the outgoing transitions from each state other than the Alpha state. A self loop transition is added to the state for each event not associated with any other outgoing transition from the state. In addition, a self loop transition is added to a state for each event associated with an outgoing transition from the state if the transition is associated with a predicate. In this case, each of the added transitions is associated with a predicate that excludes the conditions in the predicate associated with the outgoing transition. Finally, a self loop

transition is added to a state for each event associated with an outgoing transition from the state if the event can throw an exception. In this case, the predicate of the added transition is the cause for the exception to be thrown and the resulting action is throwing the exception.

Inputs: Synthesized state-based testing model that models the FIC legal behaviors and the signatures, extracted from the hook descriptions, of the FIC methods that throw exceptions.

Output: State-based testing model that models the FIC legal and illegal behaviors.

Procedure:

1. form a set s that includes all the FIC methods identifiers (i.e., method names and parameters).
2. for each state in the state transition model do
3. form a set s_1 that includes all the FIC method identifiers included in s that identify methods not invoked in the events associated with the outgoing transitions from the state
4. form a set s_2 that includes the FIC method identifiers for the methods invoked in the events associated with the outgoing transitions from the state only if the transitions have predicates. Associate with each element in the set the predicates that are associated with the corresponding transition.
5. form a set s_3 that includes the FIC method identifiers for the methods that throw exceptions and are invoked by the events associated with the outgoing transitions from the state.
6. for each element in the sets s_1, s_2, and s_3 do
   - 6.1. add a self loop transition to the state.
   - 6.2. Associate the transition with the event that invokes the method identified in the element.
   - 6.3. *if* the element is in set s_2 *then* add a predicate to the self loop transition that excludes the conditions in the predicates associated with the element.
   - 6.4. *else if* the element is in set s_3 *then* associate the cause for the exception to be thrown to the self loop transition as a predicate. In addition, associate the exception thrown to the self loop transition as an action.

Figure 3.5: Construction process of the illegal transitions of the FIC specification model.

For example, when the procedure shown in Figure 3.5 is applied to derive the illegal transitions associated with the *Frozen* state of the *NewAccount* FIC example, five self loop transitions are added to the state. Four of them are associated with the events *withdraw*, *deposit*, *freeze*, and *activate*, respectively, because these events are not associated with the outgoing transitions from the *Frozen* state. The fifth transition is associated with the *unfreeze* event because one of the outgoing transitions from the *Frozen* state is associated with the *unfreeze* event and *"balance>=0"* predicate. Therefore, the fifth added self loop transition is associated with *unfreeze* event and *"balance<0"* predicate (i.e., predicate that excludes the *"balance>=0"* condition). For the rest of the thesis, the same analysis applied for the legal transitions to generate the test cases can be applied for the illegal transitions. Adding the illegal transitions to the testing model of our concrete example makes the example more complicated and the understanding of the application of our introduced techniques on the example more difficult for the reader to follow. At the same time, adding the transitions does not introduce any additional cases not introduced for the legal transitions. Therefore, we ignored the analysis for the illegal behaviors of the *NewAccount* FIC for the rest of chapters 3 and 4.

3.2.3. Limitations

The introduced testing model synthesis technique does not guarantee synthesizing a "free of infeasible paths" model. Infeasible paths are the ones that cannot be executed. For example, in the STD of Figure 3.4, given that "amount>0" is true, the transition labeled as "7" causes several infeasible paths since depositing a positive amount of money cannot cause the balance that has a positive value to have a negative value. To solve this problem, we have to either detect the infeasible paths and avoid using them in generating the test drivers [Offutt+ 97] or we have to ignore any test driver that has violated preconditions [Cheon+ 02].

The introduced testing model synthesis technique focuses on modeling classes that have sequential behaviors. Further research is required to model classes that have

concurrent behaviors. To model such classes, synchronization contracts [Beugnard+ 99] can be used.

3.3. Generating Reusable Test Cases

This section focuses on building reusable class-based test cases for the FICs. The test cases are built using the state-based specification model constructed using the preconditions and postconditions of the FIC methods. A novel technique that overcomes the weaknesses of the existing state-based techniques is introduced. The technique works effectively in testing the FICs that do not extend the framework classes and it is extended to test the interactions between the framework classes and the FICs that extend them.

3.3.1. Testing the FICs that do not extend framework classes

Hooks can introduce FICs that do not subclass framework classes. In these FICs, compositions and existing framework components are used without inheritance. This way of customizing the framework is called black-box customization. The behavior of the FIC can be expressed in a state-transition model. Figure 3.4 shows the STD representation of a *NewAccount* banking framework interface object specification extracted from the framework hooks as illustrated in Section 3.2. The STD contains two special states: alpha and omega to represent the states of the object before being constructed and after being destroyed. Moreover, the STD contains the *Open, Overdrawn, Inactive,* and *Frozen* states to model the states of the object.

Since the state-transition model can be easily understood and is widely used in the specification-based testing techniques, we have used it in modeling the FIC specifications. In state-transition model-based testing, testers aim to achieve a certain coverage criterion. There are several state-based specification coverage techniques proposed in the literature such as all-transitions, transition-pair, full-predicate, and round-trip path coverage.

FICs are not framework classes. They are not implemented unless an application developer uses the framework hooks to implement them at the application development stage. However, since the framework hooks introduce the specifications of the FICs, the

test cases that can be used to test the FICs at the application testing stage can be produced once when the hooks are described and applied each time the FICs are used to develop an application. When any of the existing coverage techniques, except the transition-pair coverage, is applied to generate baseline test cases for FICs, only one transition sequence is required to cover a transition. For the example STD given in Figure 3.4, in the *all-transitions* technique, to cover the transition labeled 15, we can follow the path that has the sequence of transitions (1,20,15) and we do not have to worry about any other paths such as (1,12,21,15). In the transition-pair coverage, some but not all transition sequences are used to cover a transition. For example, to cover the transition-pair (15,7), we can follow the path that has the sequence of transitions (1,20,15,7) and we do not have to worry about any other paths such as (1,12,21,15,7). The sequences of transitions are used to derive the required test cases.

The application developer can decide to ignore some of the specifications for the FIC behaviors because they are unnecessary in implementing the application. Therefore, any baseline test case derived from a sequence of transitions that includes an unimplemented transition is considered broken and cannot be used as-is. Consequently, the application tester has to build new test cases or modify some baseline test cases to test the implemented transitions that were supposed to be tested using the broken baseline test cases.

For example, when the round-trip path strategy is used to derive test cases for the *NewAccount* FIC (the STD is shown in Figure 3.4), the tree shown in Figure 3.6 is constructed. Each path from the root node to a leaf node is used to build a test case. Since there are 16 such paths, 16 test cases are built. If the application developer chooses not to implement the transition originating from the *Open* state and ending at the *Inactive* state, the test cases built using the round-trip paths that include the transition are considered broken and, therefore, they cannot be used as-is. This results in breaking the test cases built from the paths that include the transition sequences labeled as (1,20,13), (1,20,15), (1,20,18), and (1,20,4). Note that the outgoing transitions from the *Inactive* state may be

implemented in the application, but none of the non-broken test cases, built using the round-trip path strategy, can test them.

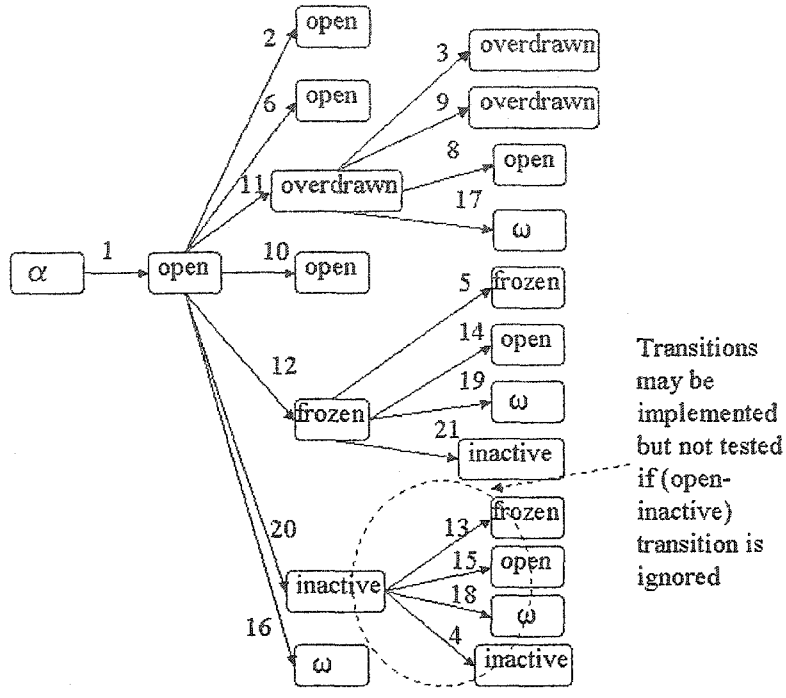

Figure 3.6: Round-trip path tree of the STD example shown in Figure 3.4.

This introduces the need for a test case generation technique that considers all sequences of transitions which can reach each state defined in the specification. A sequence of transitions form a path to a state. To solve this problem, we introduce a new coverage technique that ensures the coverage of all simple paths to each state in the state-transition model. The technique is called *all paths-state*.

In the *all paths-state* technique, we construct a set of test cases T from a specification graph SG (e.g., UML statechart or finite state machine of the FIC under test). T covers all simple paths to each state in the SG. A simple path includes only one iteration of a loop, if a loop exists in some sequence. Figure 3.7 provides a simple visualization of the idea. The coverage criterion of the technique can be written precisely as follows:

For each state in the SG, T contains tests that traverse all simple transition sequences to the state.

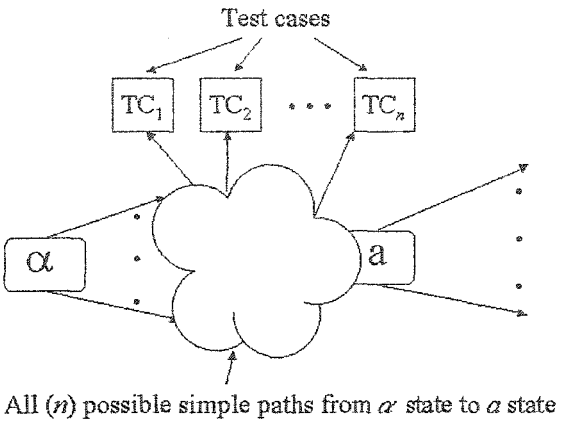

Figure 3.7: all paths-state technique idea

The set of paths that satisfy the criterion can be shown in a tree. The procedure shown in Figure 3.8 describes how to construct the tree. The procedure starts from the alpha state of the SG. In the process, whenever a state is reached the procedure traverses all the outgoing transitions from the state. The process terminates when each root-leaf tree path terminates at the omega state or a state already encountered on the path.

Figure 3.9 shows the all paths-state tree of the STD of Figure 3.4. Appendix A illustrates the steps of constructing the all paths-state tree of the STD using the procedure given in Figure 3.8. In the STD, if any transition is deleted, reachable states from the deleted transition can still be reached by some other paths of the tree. For example, if all paths-state technique is used to build the test cases and the application developer chooses not to implement the transition originated from the *Open* state and ended at the *Inactive* state, the test cases that include the transition are considered broken and, therefore, they cannot be used as-is. This results in breaking the test cases built from the paths that include the transition sequences labeled as (1,20,13,21), (1,20,13,14), (1,20,13,19), (1,20,13,5), (1,20,15), (1,20,18), and (1,20,4). Note that the remaining test cases still cover all outgoing transitions from the *Inactive* state and, therefore, can be deployed.

Test cases are generated by traversing each path in the tree from the tree root to a leaf node. The number of generated test cases is equal to the number of leaf nodes in the tree. The number of leaf nodes in the tree shown in Figure 3.9 is 22 and, therefore, the number of generated test cases is 22. Section 3.3.3 will describe how to implement the test cases.

Input: A class state-based testing model

Output: The all paths-state tree of the class model.

Procedure:

1. Draw the root node of the tree to represent the alpha state.
2. Examine the state that corresponds to each *non-terminal* leaf node in the tree and each outgoing transition from the state. At least one new edge will be drawn for each transition. Each new edge and node represents an event and resultant state reached by an outgoing transition.
   a. If the transition is unguarded, the transition guard is a simple predicate, or the transition guard is a complex predicate composed of only AND operators draw one new edge.
   b. If the transition guard is a complex predicate using one or more OR operators, draw a new edge for each truth value combination that is sufficient to make the guard TRUE.
3. For each edge and node drawn in step 2:
   a. Record the corresponding transition event, guard, and action on the new edge.
   b. If the state that the new node represents has already been encountered on the tree path that contains the new node or is the omega state, mark this node as a terminal – no more transitions are drawn from this node.
4. Repeat steps 2 and 3 until all leaf nodes are marked final.

Figure 3.8: Produce an all paths-state tree from a state model.

The following properties show the relation between the all paths-state coverage and the round-trip path coverage and the affect of deleting a transition from a state model that

has test cases built using the all paths-state technique. Each property is followed by a rationale for why the property holds.

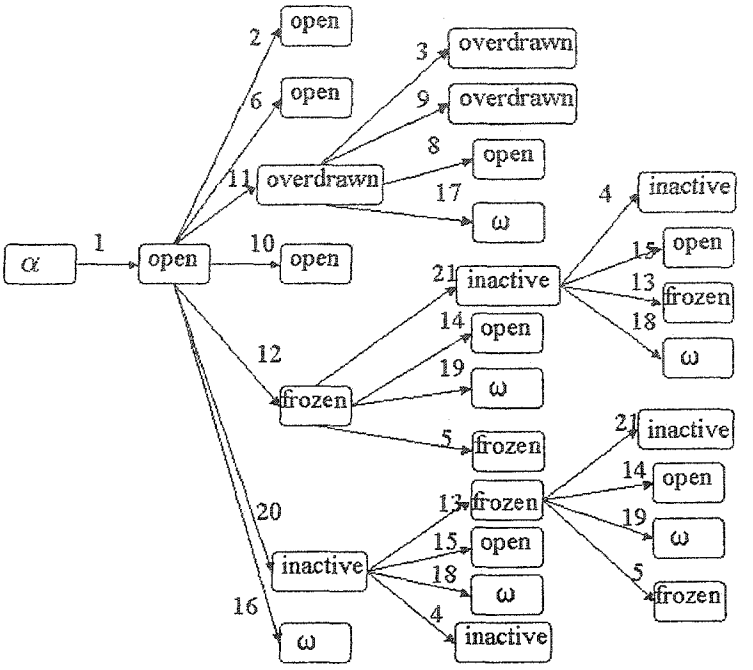

Figure 3.9: All paths-state tree, constructed using the procedure shown in Figure 3.8, of the STD example shown in Figure 3.4.

Property 3.1: *In terms of path coverage, the all paths-state coverage subsumes the round-trip path coverage.*

Rationale: The coverage of each of the all paths-state and round-trip path strategies is represented by a tree. The only difference between the construction procedures of the two types of trees is in the stopping criterion. In the round-trip path strategy, each path in the tree ends in either a node that represents the omega state in the model or a node that represents a state in the model already *represented elsewhere in the tree*. In the all-paths-state strategy, each path in the tree ends by either a node that represents the omega state in the model or a node n that represents a state in the model already *represented elsewhere in the path that contains node n*. As a result, the stopping criterion imposed by the all paths-state strategy is more constrained than the stopping criterion imposed by the round-trip path strategy. Consequently, each path in the round-trip path tree is identical to a sub-path in the all paths-state tree. Therefore, the all paths-state coverage subsumes the round-trip path coverage in terms of path coverage. Figure 3.10 shows the path coverage

hierarchy for three different strategies. The all paths-state coverage technique covers the same or more paths than the round-trip path coverage technique. The all paths-state coverage technique covers the same or less paths than the exhaustive all paths coverage criterion that covers all possible paths in a state machine.

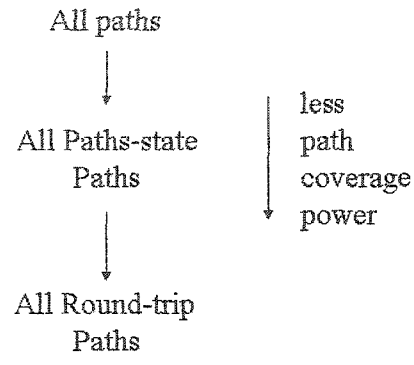

Figure 3.10: Path coverage hierarchy

Property 3.2: *When a transition in the state model is deleted, the non-broken test cases built using the all paths-state technique cover all remaining transitions in the state model initiated from the reachable states.*

Rationale: Figure 3.11 shows the different possible groups of state model paths with respect to a deleted transition. To show that the property always holds, we have to show that the remaining transitions contained in each group of paths in the modified state model are covered by the non-broken test cases as follows.

- *Groups A, B, and C*: Transitions contained in the paths of groups A, B, and C do not contain the deleted transition and, therefore, the test cases that cover them are not affected by the deleted transition. As a result, the transitions contained in the paths of groups A, B, C are covered by the non-broken test cases.

- *Group D*: Paths of Group D exist in the modified model only if paths of Group C exist. Otherwise, the model will have more than one omega state which violates the definition of the model. Therefore, if a transition is deleted and the model does not have any path of Group C, transitions contained in the paths of group D are

deleted consequently to preserve the definition of the model. On the other hand, if a transition is deleted and the model has paths of group C, the transitions contained in the paths of Group D are covered using the all paths-state technique by test cases that do not contain the deleted transition. Therefore, these test cases are not broken and they cover the transitions contained in the paths of Group D.

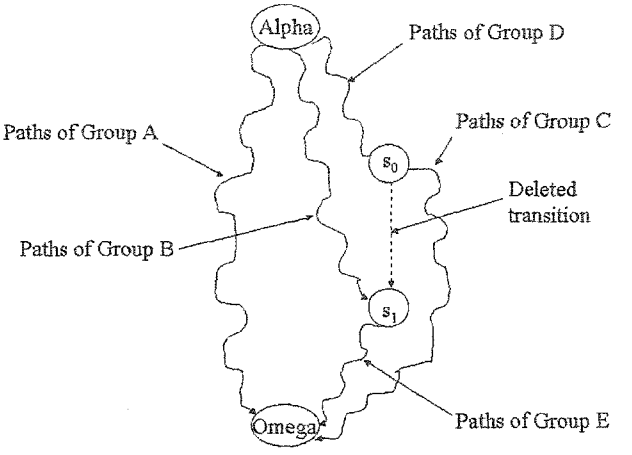

Figure 3.11: The different possible groups of state model paths with respect to a deleted transition.

- *Group E*: Paths of Group E exist in the modified model only if paths of Group B exist. Otherwise, the paths of Group E would not be reached from the alpha state, which violates the definition of the model. Therefore, if a transition is deleted and the model does not have any path of Group B, transitions contained in the paths of group E are deleted consequently to preserve the definition of the model. On the other hand, if a transition is deleted and the model has paths of group B, the transitions contained in the paths of Group E are covered using the all paths-state technique by test cases that do not contain the deleted transition. Therefore, these test cases are not broken and they cover the transitions contained in the paths of Group E.

3.3.2. Testing FICs that extend framework classes

Hooks can introduce FICs that subclass framework classes. This way of customizing the framework is called white-box customization. In this case, even if the application developer does not implement the FIC methods that override the inherited framework class methods, the inherited methods are accessible when the FIC is instantiated. Therefore, the specifications of the inherited methods defined in the hooks cannot be ignored. In terms of states and transitions, this results in having transitions that cannot be broken (i.e., must be implemented) at the application development stage. We call such transitions *guaranteed*. In Section 3.3.1, the analysis for the *NewAccount* class neglects the fact that some transitions that model the class specification are guaranteed because the *NewAccount* class extends the *Account* framework class and, therefore, the application developer cannot ignore the specifications of the *Account* class. This produced unnecessary nodes and transitions in the all paths-state tree as will be shown later in this section.

When the *all paths-state* coverage technique is applied to build test cases for the FICs that extend framework classes, some transitions may be covered using several paths. Some of the paths from the alpha state to the source state of the transition are composed of guaranteed transitions only. In this case, a path that contains only guaranteed transitions cannot be broken at the application development stage and, therefore, it is ineffective to cover the rest of the paths (i.e., paths that have not-guaranteed transitions) from alpha state to the source state of the considered transition. For example, for the STD shown in Figure 3.4, if the transitions from the *alpha* state to the *open* state and from the *open* state to the *inactive* state are guaranteed, the outgoing transitions from the *inactive* state are guaranteed to be reached by following the path of the guaranteed transitions. Since this path cannot be broken at the application development stage, there is no need to cover the other paths from the *alpha* state to the *inactive* state in the all paths-state tree to ensure the coverage of the outgoing transitions from the *inactive* state.

In the state-transition diagram, if a path to a state has all transitions marked *guaranteed*, we say that the state has a *guaranteed path*. In the all path-state tree, a state

can be represented by more than one node because we have to cover all simple paths to it such that if one path to a state has a transition not used by the application developer, the state remains reachable in the tree using other paths. *Outgoing transitions from a state that has a guaranteed path do not have to be covered in the tree using other paths*, which reduces the tree complexity. The procedure given in Figure 3.12 shows how to construct the all paths-state tree from a state-transition model that has guaranteed transitions.

The procedure starts from the root state of the state-transition model. In the process, whenever a state is reached the procedure traverses all outgoing transitions from the state. The procedure marks the tree nodes that have guaranteed paths as *guaranteed nodes*. The procedure traverses the outgoing transitions from the states represented by guaranteed nodes before the outgoing transitions from the other states. The process terminates when each root-leaf tree path ends at the omega state, a state represented previously in the path, or a state represented previously in the tree by a *guaranteed node*.

For example, suppose that the transitions that are necessary to implement the open and inactive states (i.e., transitions labeled by 1, 2, 4, 6, 10, 15, 16, 18, and 20) shown in Figure 3.4 are introduced by the *Account* class, which is a framework class. The rest of the transitions are not defined in the *Account* class, but they are defined in the hooks. In this case, the transitions labeled by 1, 2, 4, 6, 10, 15, 16, 18, and 20 are guaranteed transitions. When the procedure shown in Figure 3.12 is applied, the tree shown in Figure 3.13 is constructed. Appendix B illustrates the steps of constructing the all paths-state tree of the STD that has guaranteed transitions using the procedure given in Figure 3.12.

In Figure 3.13, the nodes that are reached by guaranteed paths are marked guaranteed. In the tree of Figure 3.9, the outgoing transitions from the node labeled by *inactive* at the end of the path labeled by (1->12->21) do not exist in Figure 3.13 because the state represented by the node labeled by *inactive* is represented elsewhere in the tree by the guaranteed node reached by the guaranteed path (1->20).

Input: A class state-based testing model that has guaranteed transitions.

Output: The all paths-state tree of the class model.

Procedure:

1. Draw the root node of the tree to represent the alpha state. Mark the node as *non-terminal* and *guaranteed.*
2. Search for a state that corresponds to a *non-terminal guaranteed* leaf node in the tree. If none is found, search for a state that corresponds to a *non-terminal not-guaranteed* leaf node in the tree.
3. Examine each outgoing transition from the state. At least one new edge will be drawn for each outgoing transition from the state. Each new edge and node represents an event and resultant state reached by an outgoing transition.
    a. If the transition is unguarded, the transition guard is a simple predicate, or the transition guard is complex predicate composed of only AND operators draw one new edge.
    b. If the transition guard is a complex predicate using one or more OR operators, draw a new branch for each truth value combination that is sufficient to make the guard TRUE.
4. For each edge and node drawn in step 3:
    a. Note the corresponding transition event, guard, action, and guarantee information on the new edge.
    b. If the edge and its source node in the tree are marked *guaranteed*, mark the destination node of the edge as *guaranteed.* Otherwise, mark it as *not-guaranteed.*
    c. If the state that the new node represents is the omega state, the state is already represented by another node (in the path containing the new node), or the state is represented somewhere else in the tree by a *guaranteed* node, mark this node as a terminal – no more transitions are drawn from this node. Otherwise, mark it as non-terminal.
5. Repeat steps 2, 3, and 4 until all leaf nodes are marked terminal.

Figure 3.12: Produce an all paths-state tree from a state model that includes guaranteed transitions.

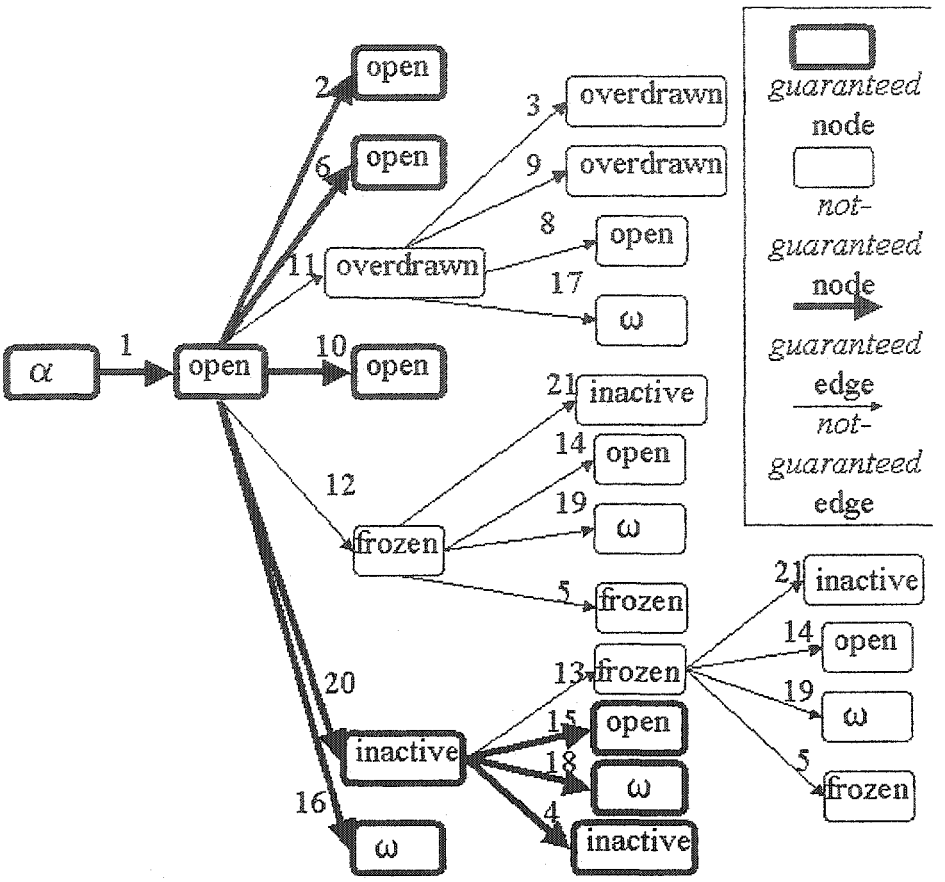

Figure 3.13: all paths-state tree, constructed using the procedure shown in Figure 3.12, of the STD example shown in Figure 3.4

The procedure given in Figure 3.14 shows how to generate the test cases from the all paths-state tree that has guaranteed nodes. The test cases are generated in two rounds. In the first round, each path from the root node to a leaf node is used to build a test case. The number of test cases built in this round is equal to the number of leaf nodes. In the second round, we search for all non-terminal nodes marked as *guaranteed* that have all outgoing edges marked as *not-guaranteed*. For each of these nodes, we build a test case that traverses the path from the root node to the node marked *guaranteed*. This round is necessary because the application developer may not use any of the methods associated with the outgoing edges from the state. In this case, all the test cases built from the paths that include the unused edges are considered broken. This results in having no test cases to test the transitions that have their destination states represented by guaranteed nodes in the tree. Figure 3.15 depicts the problem. The node labeled by *C* is a non-terminal

guaranteed node and all its outgoing edges are marked as not-guaranteed. In the first round of generating test cases, three test cases are generated to cover the paths (A->B), (A->C->D), and (A->C->E). If the application developer decides not to use the methods associated with the edges (C->D) and (C->E), the test cases generated from the paths (A->C->D), and (A->C->E) will be broken. Therefore, the edge (A->C) is not going to be covered by the remaining test case. To overcome this problem, we have introduced the second round. In the second round, the path (A->C) is used to build an additional test case.

Input: All paths-state tree that has guaranteed nodes and edges.

Output: The test cases generated from the all paths-state tree.

Procedure:

1. *for* each path from the root node to a leaf node in the all paths-state tree *do*
   Build a test case that traverses the path.
2. Search for all non-terminal nodes in the tree marked *guaranteed* and that have all their outgoing edges marked as *not-guaranteed.*
3. *for* each node *n* found in Step 2 *do*
   Build a test case that traverses the path from the root node to node *n*.

Figure 3.14: Generate test cases from the all paths-state tree that has guaranteed nodes

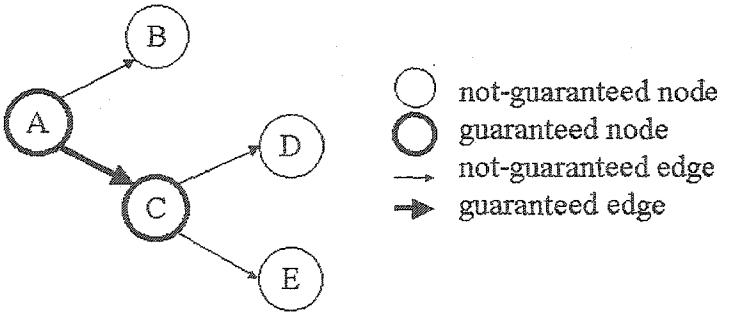

Figure 3.15: Non-terminal guaranteed node special case

3.3.3. Building test drivers

In the previous two subsections, we showed how to select the sequences of message executions (i.e., sequences of transitions that form the all paths-state paths) to be tested.

Each sequence of message executions forms a test case. To automate the testing process, it is required to generate test drivers (i.e., implementations of the test cases). To execute any message associated with a transition, it is required to set test values for the parameters of the message and to execute the code required to satisfy the predicates of the transition. We consider the general problem of the automatic generation of the test values for the parameters of the message and the automatic generation of the code required to satisfy the predicates of the transition as future work. In our work, we manually associate the required code for the test values and predicates with the transitions of the testing model.

After executing a transition, it is required to check whether the actions associated with the transition are performed correctly and whether the transition leads to the expected resulting state. Once the testing model is synthesized using the technique introduced in Section 3.2, the transition actions and the state-invariants are associated with the transitions and states of the model, respectively. When the test drivers are developed, the checking of the code for the actions and state-invariants are instrumented in the test drivers and executed at run time to evaluate the test cases.

There are two ways to implement the test cases: either to implement all of them in one class or to implement each test case in a separate class. At the application development stage, the test drivers that can be reused to test an implemented FIC are determined. These test drivers are a subset of the test drivers provided with the framework to test the FIC. If all test drivers are included in one class, the non-applicable test drivers provided with the framework would be included in the class of the test drivers and not used, which is an ineffective solution. In this thesis, we implement each test driver in a separate class. This allows the application developer to maintain only the test drivers that implement the applicable test cases instead of maintaining all the test drivers provided with the framework.

The procedure given in Figure 3.16 shows how to construct the test drivers for selected paths of a testing model. The procedure implements a class for each test case.

Each class includes a constructor method. When the constructor method is invoked at test time, the actual testing is performed. In the constructor method, the code for executing the sequence of message executions is listed. For each message associated with a transition, the code sets up the parameter values and includes the statements required to satisfy the transition predicates. The setting-up code is followed by the message invocation statement and checking statements for the resulting actions and the state-invariants of the resulting state.

In this section and for the rest of this thesis, the examples used are coded in the Java language; however, the introduced techniques are applicable for frameworks and applications written in any other object-oriented language. Figure 3.17 shows two Java test driver examples generated from the tree shown in Figure 3.9. The two test cases are generated by traversing the paths that include the transition sequences labeled as (1->2) and (1->12->14), respectively. The checking statements for the actions and the state-invariants are written as Javadoc comments using the Design-by-Contract (DbC) language [Meyer 92]. These Javadoc comments are translated at compilation time into Java code using a tool called Jcontract [Jcontract]. At run time, the Jcontract tool checks the Java statements translated from the DbC statements and reports any violations. Appendix D shows the remaining test drivers for the *NewAccount* FIC.

3.4. Summary

This chapter addresses the generation of the reusable test cases at the framework development stage. These test cases are provided with the framework to test part of the framework application whenever the framework is used at the application development stage. FICs are the application classes for which reusable test cases can be built. First, the chapter focuses on generating reusable class-based test cases for the FICs, which requires the construction of a class-based testing model. We have introduced a novel technique to synthesize the states and transitions of a class state-based testing model for the FICs from the method specifications available in the hooks.

Inputs: Paths in the FIC state-based model required to implement the test cases.

Outputs: FIC test drivers.

Procedure:

for each path required to implement a test case *do*

Create a new file

Create a class for the test driver in the file.

Create a constructor method in the class.

s is the first state in the path.

Repeat

transition t is the outgoing transition from state s in the path

if the method invoked by the transition t has parameters *then* add the code require to set the test values of the parameters to the code of the constructor method.

if the transition t has predicates *then* add the code required to satisfy the predicates to the code of the constructor method.

if the transition t is the first transition in the path *then*

insert a creation statement in the constructor method for the instance of the FIC for which the reusable test drivers are constructed.

else insert a method call statement in the constructor method for the event associated with the transition.

if the transition t has actions, insert statement(s) in the constructor method to check whether the actions associated with the transition are performed.

Insert statement(s) in the constructor method to check whether the invariants of the reached state by the transition t are satisfied.

s is the destination state of the transition t.

until state s is the last state in the path.

Figure 3.16: Construction procedure of the test drivers

Test Case # 1 (covers transition sequence 1->2)

```
public class TEST1_NewAccount{
    public TEST1_NewAccount(){
        /* testing the transition labeled as "1" */
        /* code for setting the parameter value  */
        float amount=1;
        /* invoking the message associated with the transition */
        NewAccount o = new NewAccount(amount);
        /* DbC checking statement for the invariants of the resulting state:
           Open */
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
           o.getLastActivityDate())< o.getMaxPeriod()) && !(o.isFrozen())) */

        /* testing the transition labeled as "2" */
        /* invoking the message associated with the transition */
        o.balance();
        /* DbC checking statement for the invariants of the resulting state:
          Open */
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
      o.getLastActivityDate())< o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}
```

Test Case # 12 (covers transition sequence 1->12->14)

```
public class TEST12_NewAccount{
    public TEST12_NewAccount(){
        /* testing the transition labeled as "1" */
        /* code for setting the parameter value  */
        float amount=1;
        /* invoking the message associated with the transition */
        NewAccount o = new NewAccount(amount);
        /* DbC checking statement for the invariants of the resulting state:
           Open */
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
           o.getLastActivityDate())< o.getMaxPeriod()) && !(o.isFrozen())) */

        /* testing the transition labeled as "12" */
        /* invoking the message associated with the transition */
        o.freeze();
        /* DbC checking statement for the invariants of the resulting state:
           Frozen */
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
           o.getLastActivityDate())< o.getMaxPeriod()) && (o.isFrozen())) */

        /* testing the transition labeled as "14" */
        /* invoking the message associated with the transition */
        o.unfreeze();
        /* DbC checking statement for the invariants of the resulting state:
           Open */
        /** @assert((o.balance()>=0) && ((o.getCurrebtDate()-
           o.getLastActivityDate())< o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}
```

Figure 3.17: Two test case examples generated from the tree shown in Figure 3.9

Second, we have introduced a specification coverage technique that uses the constructed state class-based testing model and produces test cases for FICs. The test cases are sufficient to cover all the implemented transitions in the specification models of the FICs under test. The introduced coverage technique is called all paths-state and it covers all paths to each state in the specification model. The coverage technique builds test cases such that when a transition in the state model is deleted, the non-broken test cases built using the coverage technique cover all remaining transitions in the state model initiated from the reachable states. Finally, the coverage technique is extended to build test cases to test the interactions between the framework classes and the FICs that extend them.

The all paths-state technique generates tests based on transition sequences and, therefore, it is useful for classes that have constraints on transition sequences. FICs are problem domain classes that often have constraints on transition sequences [Binder 99]. In contrast, the all paths-state technique may not be useful for classes that do not have constraints on the transition sequence (i.e., can accept any transition in any state). For the later classes, the test cases can be generated by analyzing the data flow in the state model [Binder 99].

Although there are several elements to be provided by the framework developer when the reusable test cases are generated, they need to be provided once and not every time an application is developed. These elements are the hook descriptions, the values of the parameters of the methods invoked in the test cases, the pieces of code required to satisfy the predicates of the transitions, and the specifications of the non-event-driven transitions. In addition, stubs used to isolate the FICs to perform class testing and testing oracles for the generated test cases are reusable. All the reusable testing elements can be provided with the framework to reduce the testing cost at the application development stage. The way of using the test cases at the application development stage has to be easy and straightforward to ensure considerable cost savings are realized when the application is tested.

Chapter 4 Using Reusable Class-Based Test Cases

4.1. Introduction

This chapter introduces an effective way to use the reusable class-based test cases. When application developers use FICs to implement their applications, they deal with the specifications of the FICs introduced by the hooks in three ways: (1) use them as defined, (2) ignore specifications for the behaviors that are unnecessary in implementing the application requirements, and (3) add new specifications for the added behaviors to meet the application requirements. This way of using the FIC specifications creates the following five main problems that have to be solved to apply the reusable test cases effectively. The problems are discussed in the subsequent sections.

1. How to find and discard the test cases for the ignored specifications.
2. How to map the names of the implemented FIC methods to the names of the FIC methods introduced in the hooks and used in the reusable test cases.
3. How to test the different implementations of the same FIC method introduced in the hooks.
4. How to deal with the flexibility that the user has in modifying the parameters of the FIC methods introduced in the hooks.
5. How to test the new specifications added by the application developers.

4.2. Tackling the Ignored Specifications Problem

Application developers have the flexibility to ignore FIC specifications introduced by the hooks if these specifications are unnecessary in implementing the application requirements. The transitions that model the ignored specifications have to be removed from the FIC state-model. The all paths-state coverage technique produces test cases such that if a transition is removed and, therefore, test cases are broken, the remaining test cases still cover the remaining used transitions. Therefore, no test cases should have to be created to test any of the reused transitions.

To find the broken test cases, it is required to associate the identifiers of the test cases with the FIC model transitions that they cover at the framework development stage. The procedure given in Figure 4.1 shows how to associate the identifiers of the test cases with the FIC transitions. Steps 1 and 2 of the procedure associate with each edge in the all paths-state tree, the identifiers of the test cases that cover the paths that contain the edge in the tree. Steps 3 and 4 associate the identifiers of the test cases with the transitions of the FIC model by mapping the edges of the tree to the transitions of the FIC model.

Inputs: The FIC state-based model, corresponding all paths-state tree, and corresponding generated test cases.

Output: FIC state-based model with test case identifiers associated with the model transitions.

Procedure:

1. Assign the empty set s of the test case identifiers to each edge in the all paths-state tree.
2. For each path in the all paths-state tree do
   2.1. Add the identifier of the test case that traverses the path to the set s of each edge in the path.
3. Assign an empty set e of the test case identifiers to each transition in the FIC testing model.
4. For each edge l in the all paths-state tree do
   4.1. Search for the corresponding transition t in the FIC testing model.
   4.2. $e_t = e_t \cup s_l$

Figure 4.1: Associating the test case identifiers to the transitions of the FIC model

In our concrete example, each path in the tree shown in Figure 3.9 is used to build a test case as discussed in Section 3.3. Table 4.1 shows the test case identifiers and the corresponding paths of the tree. Figure 4.2 shows the resulting all paths-state tree when the test case identifiers are assigned to its edges according to the first two steps of the procedure shown in Figure 4.1. For example, the edge from the node labeled *Frozen* to

the node labeled *Inactive* in the path (Alpha->Open->Frozen->Inactive) in the tree shown in Figure 4.1 is associated with the test case identifiers 8, 9, 10, and 11 because the edge is contained in the tree paths covered by these test cases. Figure 4.3 shows the resulting STD of the *NewAccount* FIC when the test case identifiers are assigned to the transitions according to steps 3 and 4 of the procedure shown in Figure 4.1. For example, the transition from *Frozen* to ω states is represented twice in the all paths-state tree shown in Figure 4.2 by edges covered by the test cases that have identifiers 13 and 17, respectively. Therefore, the transition is associated with a set of test case identifiers {13,17}. The test case identifiers associated to each transition are shown also in the fifth column of Table 4.2.

Test case identifier	**Path**
1	α, open, open
2	α, open, open
3	α, open, overdrawn, overdrawn
4	α, open, overdrawn, overdrawn
5	α, open, overdrawn, open
6	α, open, overdrawn, ω
7	α, open, open
8	α, open, frozen, inactive, inactive
9	α, open, frozen, inactive, open
10	α, open, frozen, inactive, frozen
11	α, open, frozen, inactive, ω
12	α, open, frozen, open
13	α, open, frozen, ω
14	α, open, frozen, frozen
15	α, open, inactive, frozen, inactive
16	α, open, inactive, frozen, open
17	α, open, inactive, frozen, ω
18	α, open, inactive, frozen, frozen
19	α, open, inactive, open
20	α, open, inactive, ω
21	α, open, inactive, inactive
22	α, open, ω

Table 4.1: identifiers of the test cases built from the paths of the tree of Figure 3.9.

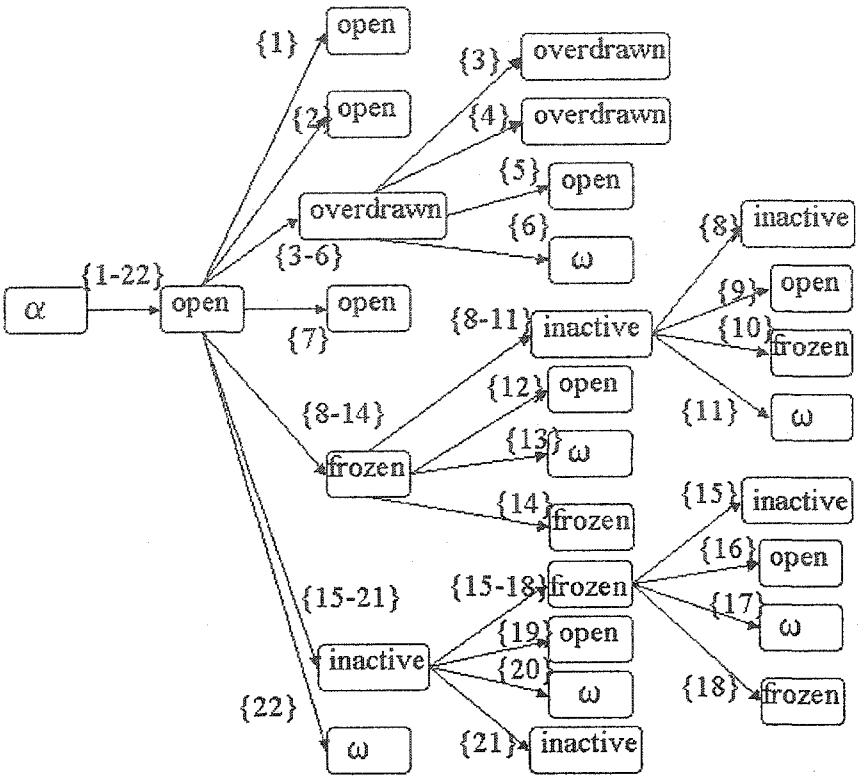

Figure 4.2: All paths-state tree of the *NewAccount* FIC with identifiers of the test cases assigned to the edges

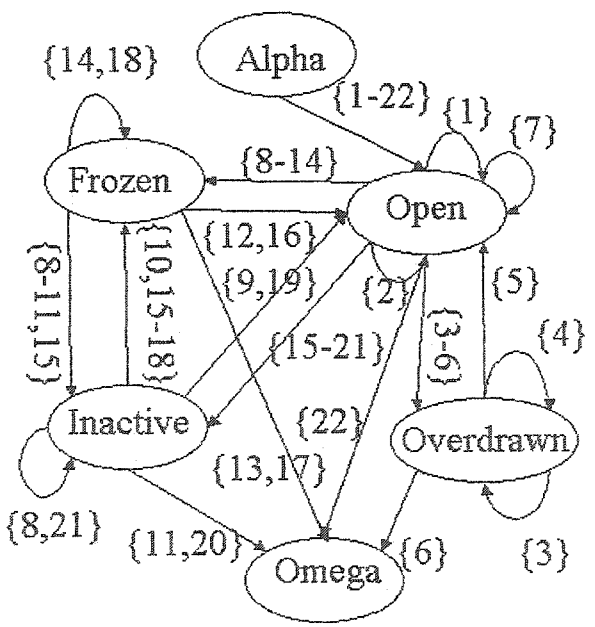

Figure 4.3: The STD of Figure 3.4 with identifiers of the test cases assigned to the transitions

Transition identifier	Event	Source state	Destination state	Test cases identifiers	Test cases identifiers after removing (frozen,inactive) transition
1	NewAccount	α	open	1-22	1-7,12-14,16-22
2	balance	open	open	1	1
3	balance	overdrawn	overdrawn	3	3
4	balance	inactive	inactive	8,21	21
5	balance	frozen	frozen	14,18	14,18
6	deposit	open	open	2	2
8	deposit	overdrawn	open	5	5
9	deposit	overdrawn	overdrawn	4	4
10	withdraw	open	open	7	7
11	withdraw	open	overdrawn	3-6	3-6
12	freeze	open	frozen	8-14	12-14
13	freeze	inactive	frozen	10,15-18	16-18
14	unfreeze	frozen	open	12,16	12,16
15	activate	inactive	open	9,19	19
16		open	ω	22	22
17		overdrawn	ω	6	6
18		inactive	ω	11,20	20
19		frozen	ω	13,17	13,17
20		open	inactive	15-21	16-21
21		frozen	inactive	8-11,15	

Table 4.2: Test case identifiers associated with the transitions of the STD of Figure 3.4.

At the application development stage, when the application developer ignores FIC specifications, the transitions in the testing model corresponding to the ignored specifications are removed. In addition, the transitions no longer contained in any path from Alpha to Omega states are removed. The broken test cases are the ones whose identifiers are associated with the removed transitions.

Let us look at an example. Suppose the application developer decides to remove the transition that has identifier 21 in Table 4.2. From the fifth column of Table 4.2 we see that the test cases that have identifiers 8, 9, 10, 11, and 15 cover the deleted transition and, therefore, they are broken. The identifiers of the broken test cases have to be

removed from all cells in the fifth column of Table 4.2. The results of these removals are shown in the last column of Table 4.2. The table shows that each of the remaining transitions is covered by at least one test case.

In conclusion, in order to detect the broken test cases it is necessary to provide the framework with the FIC models that have test case identifiers associated with their transitions.

4.3. Tackling the Renaming Problem

One of the problems in reusing the test cases is that the test cases use the method names introduced by the hooks, while the actual implementation to be tested uses method names introduced by the application developer. For example, when the application developer of the banking system framework implements the *NewAccount* class he can rename it *MyAccount* and change some of the names of the class methods as shown in Table 4.3. Test cases generated at the framework development stage would not use the new class and method names and, therefore, could not be reused as-is directly.

Method declaration in Banking System framework hooks	**Method declaration in MyAccount class**
NewAccount(float)	MyAccount(float)
deposit(float)	USdeposit(float) EURdeposit(float)
withdraw(float)	withdraw100() withdraw(float)
balance()	getBalance()
freeze()	freeze()
unfreeze()	unfreeze()
activate()	activate()

Table 4.3: Method-name-mapping table for the *MyAccount* class

To solve this problem, a mapping class that has the same name as the FIC class defined in the hooks has to be built. The mapping class inherits the implemented class (e.g., the *MyAccount* class) and its methods map the methods introduced by the

framework hooks to the ones used in the actual implementation of the class. The mapping is achieved by using a method-name-mapping table as illustrated in Table 4.3. Given the method-name-mapping table, the generation of the mapping class is straightforward and can be easily automated: whenever a method listed in the first column of the method-name-mapping table is invoked by a test driver (i.e., implementation of a test case), the invoked method in the mapping class invokes the corresponding method listed in the second column of the table. For example, Figure 4.4 shows the *NewAccount* mapping class that uses the method-name-mapping table shown in Table 4.3.

```
public class NewAccount extends MyAccount {
        public NewAccount(float amount) {
                super(amount);
        }
        public float balance() {
                return getBalance();
        }
        public void deposit(float amount) {
                switch((new DRIVER_MyAccount()).getSwitchkey()) {
                        case 1:USdeposite(amount);
                                break;
                        case 2:EURdeposit(amount);
                                break;
                }
        }
        public void withdraw(float amount)  {
                switch((new DRIVER_MyAccount()).getSwitchkey()) {
                        case 1:super.withdraw(amount);
                                break;
                        case 2: super.withdraw100();
                                break;
                }

        }
        public void freeze()  {
                super.freeze();
        }
        public void unfreeze()  {
                super.unfreeze();
        }
        public void activate()  {
                super.activate();
        }
}
```

Figure 4.4: *NewAccount* mapping class

In the *MyAccount* class, the constructor method is renamed to match the name of the new class name as shown in Table 4.3. Therefore, when test drivers call the *NewAccount* constructor method, the *MyAccount* constructor method is called. In Java, the renaming

problem is not a problem for the constructor methods, because the constructor method of the superclass is always invoked using the *super* keyword regardless of the superclass name. However, the problem has to be solved as illustrated above when methods other than the constructor method are renamed. For example, the *balance* method is renamed *getBalance* in the *MyAccount* class. When test drivers call the *balance* method of the *NewAccount* class, the method invokes the *getBalance* method as shown in Figure 4.4.

The mapping class is useful also when the application developer does not implement an instance variable introduced in a hook description and the variable access method. If the access method of the instance variable is used in the reusable test drivers to check the state-invariants and not contained in the implemented FIC, the reusable test drivers would not compile. In this case, the access method of the instance variable has to be implemented in the mapping class. The method returns any value accepted at any of the original FIC model states remaining in the modified model. For example, suppose the application developer who uses the *NewAccount* FIC does not implement the *frozen* instance variable and its access method *isFrozen()*. This causes the *Frozen* state and the transitions associated to it to be removed from the STD shown in Figure 3.4. Despite the fact that *Test Case # 1* shown in Figure 3.17 does not cover any of the removed transitions, it does not compile because it uses the *isFrozen* method not contained in the implemented FIC to check the state-invariants. To solve this problem, the mapping class should implement the *isFrozen* method as follows:

public boolean isFrozen(){ return false; }

The method returns *false* because *false* is the accepted value of the *frozen* instance variable in the STD states shown in Figure 3.4 and remaining in the model that represents the implemented FIC.

In conclusion, we can reuse the test drivers generated at the framework development stage as-is (i.e., without modifying them) by using a mapping class which maps the methods invoked in the test drivers to the methods used in the actual implementation of

the FIC. In addition, the mapping class implements access methods used in the test drivers to check the state-invariants and not contained in the implemented FIC.

4.4. Tackling the Different Implementations of a FIC Method Problem

In some cases, the application developer can decide to have different implementations for a method introduced by the hooks. For example, suppose that the application developer of the *MyAccount* class decides to have two implementations for the *deposit* method introduced by the banking framework hooks: one for depositing US money and the other one for depositing EUR money. These different implementations have common preconditions and postconditions introduced by the hooks because they are constructed using the same hooks. The different implementations can have the same method name but different parameters, or they can have the same parameters but different method names.

To test the different implementations, the test drivers that test the method should be exercised as many times as the number of implemented versions of the method. To do so, a *SwitchKey* global variable accessed by both the mapping class and the class that invokes the test drivers is used to keep track of the order of the version to be called when the test drivers are exercised.

As an example, suppose that the application developer of the *MyAccount* class implements two versions of the *deposit* method as indicated earlier in Table 4.3. Each version has a different method name. However, both versions have common preconditions and postconditions provided in the hooks for the *deposit* method and, therefore, the reusable test cases generated for the *deposit* method have to be applied for both implemented versions. Thus, the following code is included in the *NewAccount* class as shown in Figure 4.4:

```
public void deposit(amount) {
    switch((new DRIVER_MyAccount).getSwitchkey()) {
        case 1:USdeposit(amount);
            break;
        case 2:EURdeposit(amount);
```

```
                    break;
            }
    }
```

In the above code, the return value of the *getSwitchKey* method is used to determine at run time which *deposit* method implementation to invoke. The *getSwitchKey* method is defined in the driver class that invokes the test drivers as will be illustrated in Section 4.7.

This way of testing the different implementations of the same FIC method allows for reusing the test drivers generated at the framework development stage as-is to test the different implementations of the FIC methods. No test drivers have to be created from scratch to solve the problem, thereby reducing the application class testing time.

4.5. Tackling the Method Parameter Update Problem

Application developers have the flexibility to add or remove parameters from the parameter list of the FIC methods introduced by the hooks as long as they do not change the preconditions and postconditions introduced in the hooks. When an application developer removes one or more parameters from the implemented version of the method introduced by a hook, the unused parameters are just ignored at the time the test drivers invoke the method introduced by the hook. As an example, suppose that the *MyAccount* class is implemented. In the implemented version of the *withdraw* method, the parameter of the withdraw method introduced by the hook is removed to restrict withdrawals to only a fixed amount of money. In the implementation of the *withdraw100* method, the application developer decides to pass the parameter value hard-coded to the *super.withdraw* method as follows:

```
public class MyAccount extends Account {
    ...
    public void withdraw100() {
        super.withdraw(100);
        ...         }
    ...
}
```

In this case, as shown in Figure 4.4, the *NewAccount* mapping class ignores the parameter value passed to the *withdraw(float)* method of the class when the *withdraw100()* method is invoked.

When the application developer adds more parameters to the parameter list of a method introduced by a hook, the application developer has to pass a hard-coded value to the added parameters when the method is invoked in the class that inherits the implemented class. The application developer has to determine the values to be passed to such parameters. If more than one test value has to be exercised, the application developer has to find the test drivers that invoke the method and execute them with the other test values of the parameter.

In conclusion, instead of modifying the reusable test drivers, the application developer can use the mapping class to solve the method parameter update problem, which reduces the cost of using the test drivers at the application development stage. However, in some cases, changing the parameters of the methods introduced by the hooks implies adding more constraint preconditions. In such cases, the application developer has to add manually these more constraint preconditions as predicates to the transitions that invoke the methods.

4.6. Tackling the Test Case Augmentation Problem

To develop an application using a framework, the application developers implement the FICs. The specifications of the implemented FICs have two sources: (1) the specifications introduced at the framework development stage in the hooks and (2) the specifications of the new methods added to the FICs. The latter specifications are added at the application development stage and, therefore, have no reusable test cases to cover them. Covering these specifications may require augmenting reusable test cases or creating new test cases from scratch.

The placement of the transitions that model the added specifications determine whether covering the added specifications requires augmenting reusable test cases or creating test cases from scratch. If alpha is the source state of an added transition or all the paths from the alpha state to the source state of the added transition consist of only added transitions (i.e., as shown in Figure 4.5(b)), covering the added transition requires creating a test case from scratch. Otherwise (i.e., as shown in Figure 4.5(a)), covering the added transition can be achieved by augmenting a reusable test case.

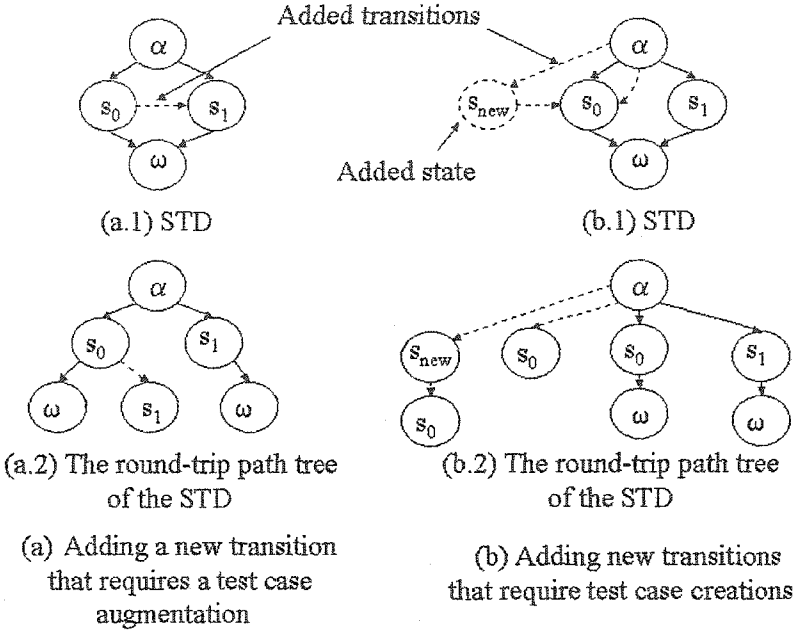

Figure 4.5: Possible placements of the added transitions

The procedure given in Figure 4.6 illustrates how to augment and create test cases to cover the added transitions. In this procedure, a round-trip path tree [Binder 99] is built as illustrated in the procedure given in Figure 4.7 for the modified state-transition model (i.e., the model that contains the added transition) and the non-broken test case identifiers associated with the model transitions are associated with the corresponding tree edges. The procedure either builds new test cases or augments reusable test cases to cover the added transitions represented in the tree by edges not associated with any test case identifiers.

Inputs: The state-based model of the implemented FIC and the non-broken test cases.
Outputs: Test cases that cover the added transitions.
Procedure:

1. Build a round-trip path tree for the specification state-transition model of the implemented FIC according to the procedure given in Figure 4.7.
2. Associate the non-broken test case identifiers to the edges of the round-trip path tree as illustrated in steps 1 and 2 of the procedure given in Figure 4.1.
3. Pick an edge *l* not associated with any test case identifier
4. Count the number *np* of the root-leaf tree paths that contain the edge *l*.
5. If the source node of the edge is reached by an edge *k* with which test case identifiers are associated then
   5.1. Pick one of the test cases that its identifier is associated with the edge *k*.
   5.2. Make *np* copies of the test case and give each of them different identifier.
   5.3. for each copy *t* of the test cases do
      5.3.1. Give the test case a new identifier.
      5.3.2. In the implementation of the test case (i.e., test driver), search for the statement *s* corresponding to the event associated with the edge *k*.
      5.3.3. Remove all invocation statements after the statement *s*.
      5.3.4. Select one of the *np* tree paths that some of its edges are not associated with test case identifiers.
      5.3.5. Associate the edges of the selected tree path with the new identifier of the test case *t*.
      5.3.6. Insert the invocation statements for the events associated with the edges that follow the edge *k* in the selected tree path.
6. else if the edge *l* is initiated from the root node of the tree do
   6.1. Create *np* test cases from scratch. Each test case covers one of the *np* tree paths that contain the edge *l*.
   6.2. Associate the edges of the paths with the identifiers of the test cases that cover the edges.
7. Repeat step 3, 4, 5, and 6 until each of the edges in the tree is associated with at least one test case identifier.

Figure 4.6: Constructing test cases to cover added transitions

If the source node of the edge *k* not associated with test case identifiers is reached by an edge associated with a test case identifier (i.e., as shown in Figure 4.5(a)), the test case is augmented to cover the added transition represented by the edge *k*. Otherwise, if the source node of the edge represents the alpha state (i.e., as shown in Figure 4.5(b)), a test case is created from scratch to cover the added transition represented by the edge *k*. Whenever a test case is augmented or created from scratch, the tree edges that represent the transitions covered by the test case are associated with the identifier of the test case. The procedure terminates when each of the edges of the tree is associated with at least one test case identifier.

Input: A class state-based testing model
Output: The round-trip path tree of the class model.
Procedure:

1. The initial state is the root node of the tree. Use the alpha state if multiple constructors produce behaviorally different initial states.
2. Search for a state that corresponds to *non-terminal* leaf node in the tree.
3. Examine each outgoing transition from the state. At least one new edge will be drawn for each outgoing transition from the state. Each new edge and node represents an event and resultant state reached by an outgoing transition.
   a. If the transition is unguarded, draw one new branch.
   b. If the transition guard is a simple predicate or a complex predicate composed of only AND operators, draw one new branch.
   c. If the transition guard is a complex predicate using one or more OR operators, draw a new branch for each truth value combination that is sufficient to make the guard TRUE.
4. For each edge and node drawn in Step 2:
   a. Note the corresponding transition event, guard, and action information on the new edge.
   b. If the state that the new node represents is a final state or the state is represented somewhere else in the tree, mark this node as a terminal – no more transitions are drawn from this node. Otherwise, mark it as non-terminal.
5. Repeat steps 2, 3, and 4 until all leaf nodes are marked terminal.

Figure 4.7: Constructing round-trip path tree [Binder 99]

Let us look at an example. Suppose the application developer decides to develop the *MyAccount* FIC and to add a method that inactivates an overdrawn account. This new specification is modeled in the STD of Figure 3.4 by a transition from the *overdrawn* state to the *inactive* state. Following steps 1 and 2 of the procedure given in Figure 4.6, as shown in Figure 4.8, we construct the round-trip path tree of the modified model and associate the edges of the tree with test case identifiers according to the STD shown in Figure 4.3. In Step 3 of the procedure, we pick the edge that represents the added transition because it is not associated with any test case identifier. According to Step 4, the number of paths that contain the edge is one. The source node of the edge is reached by an edge k associated with test case identifiers: 3, 4, 5, and 6. In steps 5.1 and 5.2, we pick the test case identifier 3 and make a copy of it to be augmented. Figure 4.9 shows the Java implementation of the test case. The augmented version of the test case is shown in Figure 4.10. The identifier of the augmented test case is 23. According to Step 5.3.2, the statement that corresponds to the event associated with the edge k is in line 13 of the code listed in Figure 4.9 and, therefore, we apply Step 5.3.3 to remove any code statement after line 13 (i.e., statements in lines 17-21). In steps 5.3.4 and 5.3.5 of the procedure, we associate all the edges in the tree path that contains the edge that represents the added transition with the test case identifier 23 (i.e., the identifier of the augmented version of the test case). Finally, according to Step 5.3.6, we insert the invocation statement of the inactivate method as shown in line 22 of the code listed in Figure 4.10.

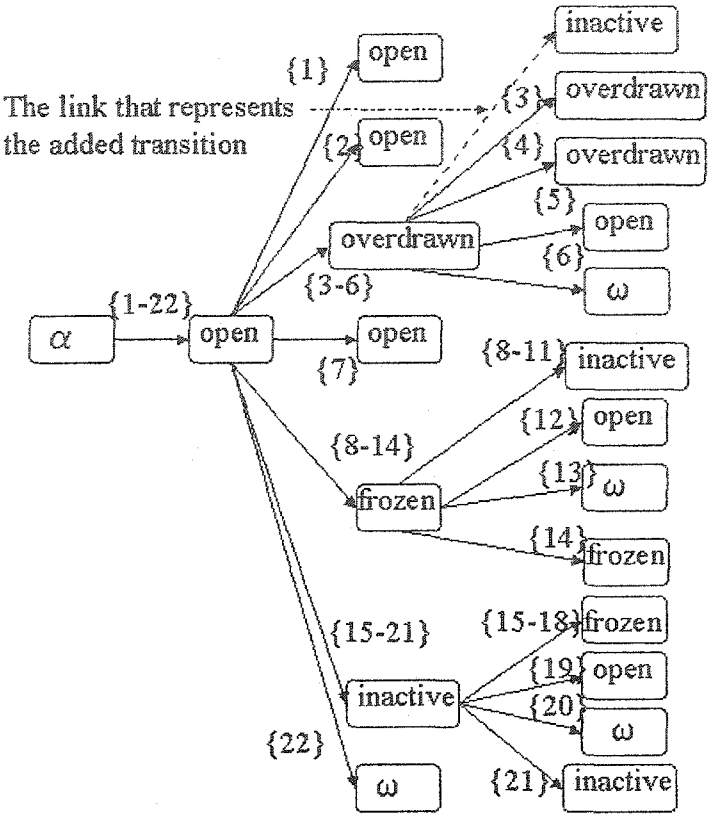

Figure 4.8: The round-trip path tree of the modified version of the STD shown in Figure 4.3

```
1  public class TEST3_NewAccount{
2      public TEST3_NewAccount(){
3          /* Test transition: source state: Alpha, sink state: Open, event:
4             NewAccount(amount), predicates: amount>=0 */
5          float amount=1;
6          NewAccount o = new NewAccount(amount);
7          /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
8            o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen()))*/
9
10         /* Test transition: source state: Open, sink state: Overdrawn,
11            event: withdraw(amount), predicates: (balance - amount)<0 */
12         amount=1+o.balance();
13         o.withdraw(amount);
14         /** @assert((o.balance()<0) && ((o.getCurrentDate()-
15           o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen()))*/
16
17         /* Test transition: source state: Overdrawn,sink state: Overdrawn,
18            event: balance(), predicates: none */
19         o.balance();
20         /** @assert((o.balance()<0) && ((o.getCurrentDate()-
21           o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen()))*/
22     }
23 }
```

Figure 4.9: The Java implementation of the test case that has identifier 3

```
1  public class TEST23_NewAccount{
2      public TEST23_NewAccount(){
3          /* Test transition: source state: Alpha, sink state: Open, event:
4             NewAccount(amount), predicates: amount>=0 */
5          float amount=1;
6          NewAccount o = new NewAccount(amount);
7          /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
8            o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen()))*/
9
10         /* Test transition: source state: Open, sink state: Overdrawn,
11            event: withdraw(amount), predicates: (balance - amount)<0 */
12         amount=1+o.balance();
13         o.withdraw(amount);
14         /** @assert((o.balance()<0) && ((o.getCurrentDate()-
15           o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen()))*/
16
17         /* The following code implements the difference between the test
18          * driver shown in Figure 4.9 and the augmented version of the
19          * test driver */
29         /* Test transition: source state: Overdrawn,sink state: Overdrawn,
21            event: inactivate(), predicates: none */
22         o.inactivate();
23         /** @assert((o.balance()<0) && ((o.getCurrentDate()-
24           o.getLastActivityDate())>=o.getMaxPeriod())&& !(o.isFrozen()))*/
25     }
26 }
```

Figure 4.10: The augmented version of the implementation of the test case shown in Figure 4.9

4.7. Invoking Test Drivers

Finally, it is required to build a driver class for each implemented FIC to invoke the non-broken reused as-is, augmented, and new test drivers that test the FIC. If the *switchKey* global variable is required to allow for testing the different implementations of a FIC method as illustrated in Section 4.4, the variable is defined as a global variable in the driver class. In Java for example, the *switchKey* variable is declared private and static and an access method is implemented to get the variable value. For example, part of the driver class for the *MyAccount* test drivers is shown in Figure 4.11. As indicated in Section 4.4, the *deposit* method introduced in the hooks has two different implementations in the *MyAccount* FIC. Therefore, as shown in Figure 4.11, the driver class declares the *switchKey* instance variable and its access method (i.e., *getSwitchKey()*). From the second and fifth columns of Table 4.2, it is found that *deposit* method is covered using the test cases that have identifiers 2, 4, and 5. As depicted in Figure 4.11, the test cases that have identifiers 2, 4, and 5 are exercised twice: once after

the value of the *switchKey* variable is set to "1" and once after the value of the variable is set to "2".

```
public class DRIVER_MyAccount{
     private static int switchKey=1;
     public int getSwitchKey() {
          return switchKey;
    }
     public static void main(String args[]){
          …
          /* switchKey is already set to 1  */
          /* Invoking the test drivers that cover the first
             implementation of the deposit method  */
          new TEST2_NewAccount();
          …
          new TEST4_NewAccount();
          new TEST5_NewAccount();
          …
          switchKey=2;
          /* Invoking the test drivers that cover the second
             implementation of the deposit method  */
          new TEST2_NewAccount();
          new TEST4_NewAccount();
          new TEST5_NewAccount();
          …
     }
}
```

Figure 4.11: Part of the DRIVER_MyAccount class

Figure 4.12 shows the class diagram that represents the relation between the implemented FIC under test, the mapping class, the test drivers, and the driver class. The mapping class extends the FIC under test and the test drivers depend on the mapping class. Finally, the driver class depends on the test drivers. After implementing the driver class, the driver has to be executed to perform the actual testing.

4.8. Fault Detection

The reusable test cases are generated using the all paths-state technique. As stated in Property 3.1, the all paths-state coverage subsumes the round-trip path coverage in terms of state machine path coverage. At the application development stage, some of the reusable test cases are broken and cannot be used. Other reusable test cases are either used as-is or augmented. In some cases, new test cases are created from scratch to test new specifications. The following property compares the fault coverage of the resulting test cases applied at the application development stage to test the implemented FICs with

the fault coverage of the round-trip path test cases. In [Antoniol+ 02], it is shown that the round-trip path test cases are reasonably effective in terms of fault coverage (i.e., 87% average fault coverage). FICs are problem domain classes, which are often suitable for testing with the round-trip path technique [Binder 99].

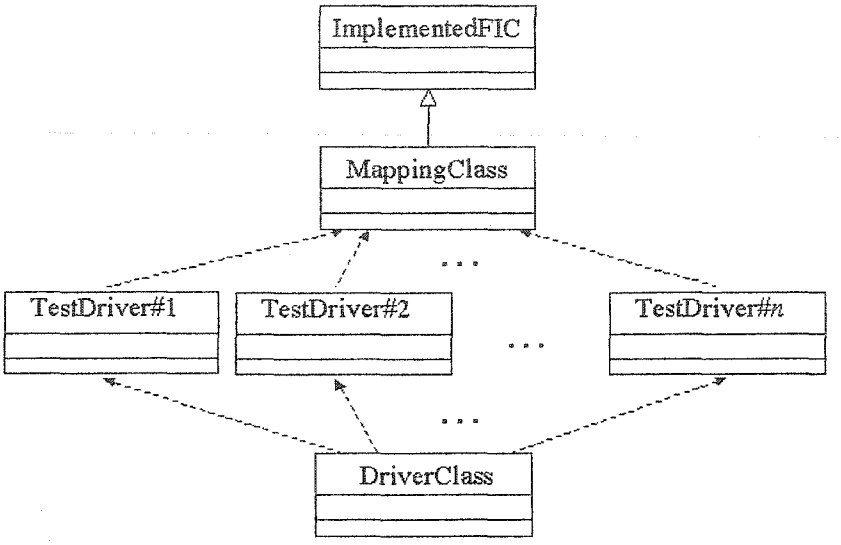

Figure 4.12: The class diagram of the FIC under test and the classes required in testing

Property 4.1: *After removing the broken test cases, augmenting some test cases, using some test cases as-is, and creating some test cases from scratch, the resulting test cases applied at the application development stage to test the implemented FICs have at least the same fault coverage as the round-trip path test cases.*

Rationale: By definition, the all paths-state tree covers all simple transition sequences to each state in the state model. When a transition is deleted, the paths that include it are broken. Therefore, the rest of the paths in the resulting tree cover all simple transition sequences to each state in the state model except for the sequences that include the deleted transition. This means that the resulting tree covers all simple transition sequences to each state in the updated state model and, therefore, it is an all paths-state tree. Property 3.1 states that all paths-state coverage subsumes the round-trip path coverage in terms of path coverage. Therefore, for the used as-is transitions, the non-broken test cases covered by the resulting all paths-state tree have at least the same fault coverage as the round-trip path test cases that cover the reused transitions.

The transitions added to the state model because of the new code added by the application developer are covered using round-trip path strategy as illustrated in Section 4.6. As a result, after removing the broken test cases, augmenting some test cases, using some test cases as-is, and creating some test cases from scratch, the resulting test cases applied at the application development stage to test the implemented FICs have at least the same fault coverage as the round-trip path test cases.

4.9. Summary

This chapter demonstrated how to use the reusable test cases by effectively addressing five problems that could prevent test case reusability. The chapter also demonstrated how to build a driver for the reusable test cases and showed that the resulting test cases have at least the same fault coverage as the round-trip path test cases. The proposed solutions fully automate the use of the test cases with the exception of the case involving added parameters introduced by the application developer to the methods introduced by the hooks. In this case, the application developer has to provide test data for the added parameters.

All paths-state coverage technique builds redundant test cases in the sense that all transitions covered by a test case can be covered by other test cases. For example, as shown in Table 4.2, the test case number 15 covers the transitions that have identifiers 1, 13, 20, and 21. These transitions are covered by other test cases such as 1, 10, 16, and 8, respectively. However, none of the redundant test cases are identical in the sense that the test cases that cover the same transitions exercise the transitions using different paths. Therefore, the redundant reusable test cases generated using all paths-state technique have different fault detection capabilities and none of them should be removed at the application development stage except for the broken ones.

Chapter 5 Case Studies

This chapter describes experimental studies of the relationship between the type of the framework and the use of the FICs in the framework applications to assist the framework developer in deciding whether to build reusable test cases for the FICs or not. Moreover, the case studies evaluate the proposed techniques with regard to test case reusability and specification coverage in terms of the number of transitions in the state model. In the evaluation, the practicality of the proposed test case generation and use technique is shown using several examples. Finally, an empirical evaluation is conducted to study the degree of coverage for the specifications of the implemented FICs, in terms of the number of transitions in the state model, achieved using test cases generated using the all paths-state technique comparing to test cases generated using the round-trip path and the all-transitions techniques.

Fifteen applications developed using the following four frameworks were considered in the case studies: Client-Server Framework (CSF) [CSF], Swing [Swing], SalesPoint [SalesPoint], and WaveFront pattern frameworks [WaveFront Documentation]. CSF and Swing are application frameworks, while SalesPoint and WaveFront pattern frameworks are domain frameworks.

CSF is a communications framework written in Java and developed to support the building of relatively small applications that require client-server or peer-to-peer communication support. CSF also provides persistent storage capabilities and can handle communications over a TCP/IP connection using a model similar to email. CSF deals with synchronous and asynchronous messages sent between remote objects. The framework code consists of 38 classes and about 1.4K lines of code (without comments/blank lines). CSF hooks describe the behavior of ten FICs and show how they can be implemented or customized.

Swing is a Java framework developed to support GUI applications. In Java 1.3.1, Swing consists of 460 classes but no documented hooks are available. As we will explain

later in Section 5.1.2, the FICs for this framework were derived from an analysis of the application set used in the first case study.

SalesPoint is a framework written in Java and developed to create point-of-sale simulation applications such as a ticket vending machine application or a big supermarket with many departments application. The framework supports the management of the relations between the business, the customers, and the administrative tasks like accounting. The SalesPoint framework consists of 161 classes and it is provided with hooks that describe the behavior of 78 FICs.

WaveFront pattern frameworks support the computation of dependent elements. The frameworks are generated from the WaveFront Pattern (WFP) using the CO_2P_3S parallel programming system [McDonald+ 97]. Two relatively small WaveFront pattern frameworks are considered. Each of them consists of six classes and about 200 lines of Java code (without comments/blank lines). For each of them, three hooks were provided to document the way of using the framework. The hooks of each framework describe the behavior of two FICs and show how they can be implemented or customized.

5.1. Case Study 1: Generating Reusable Test Cases For Framework Applications: Is It Worth It?

5.1.1. Introduction

In this section, we measure the application class testing cost reduction using the reusable test cases generated at the framework development stage. The cost reduction is measured in terms of the number of implemented FICs in the applications and their total number of lines of code (LOC) in comparison to the total number of classes implemented at the application development stage and their total number of lines of code (LOC). We count the number of LOC because it is a commonly used measurement for the size of code.

The case study was conducted using thirteen randomly selected applications developed using three frameworks: CSF, Swing, and SalesPoint. The former two frameworks are application frameworks while the latter one is a domain framework. The WaveFront frameworks are not used in this case study because their applications have relatively small number of classes and, therefore, the results of applying the case study analysis on them are not expected to be representative.

The results of the case study show that the percentage of use of the FICs in the applications constructed using the domain framework is, on average, much higher than the percentage of use of the FICs in the applications built using the application frameworks. As a result, the reduction in class testing cost of the applications constructed using the domain frameworks is, on average, much higher than the reduction in class testing cost of the applications built using the application frameworks.

5.1.2. Case study set-up

Performing the analysis required in this case study for relatively large number of applications requires exhaustive effort. Therefore, the case study was conducted using thirteen randomly selected applications out of a pool of 39 applications. Five of the applications use one framework and eight applications use two frameworks. As shown in tables 5.1-5.4, the applications use the following frameworks: one application uses CSF only, two applications use the Swing framework only, two applications use the SalesPoint framework only, four applications use CSF and the Swing framework, and four applications use the SalesPoint and Swing frameworks. The contents of these tables will be explained in greater detail in the next two subsections.

The CSF applications were developed by fourth-year undergraduate students at the University of Alberta. The SalesPoint framework applications were developed by second-year undergraduate students at the University of the Federal Armed Forces Munich. Finally the Swing applications were developed by a combination of the second and fourth-year undergraduate students, in conjunction with their application development activities on CSF and SalePoint.

For each application, the classes implemented at the application development stage were counted. The number of classes does not include the number of framework classes. In addition, the number of LOC of the counted classes is also counted. These two figures were counted using the LOCC tool [LOCC]. LOCC is a Java tool that produces size data corresponding to the number of packages, the number of classes in each package, the number of methods in each class, and the number of lines of code in each package, class, and method. The LOCC tool does not count comments and blank lines as part of the lines of code. Tables 5.1, 5.2, and 5.3 show the application name, the total number of application classes not including framework classes, and the number of lines of code (LOC) of each application. The FICs included in the applications were counted manually. Since the Swing framework has no associated hooks, we used the definition of FIC stated in Section 1.2 to find the implemented FICs and count them. Every class in the considered applications that extends or uses a Swing class is an implemented FIC. For each application developed using the Swing framework, we counted the implemented Swing FICs. Finally, the total number of LOC for the FICs is the summation of the LOC of each of the FICs counted using LOCC tool.

5.1.3. Case study results

For each application, the first column of tables 5.1, 5.2, and 5.3 shows the name of the application. The second column shows the number of application classes not including the framework classes. The third column shows the number of LOC of the classes counted in the second column. The fourth column shows the number of FICs implemented in the application and the percentage of the number of FICs in the application. The last column shows the total number of LOC of FICs and the percentage of the number of LOC of the FICs in the application.

For applications developed using application frameworks, Table 5.1 shows that an average of 41.4% of the classes of the CSF applications are FICs. In terms of LOC, an average of 28.3% of the LOC of the CSF applications are for FICs. Table 5.2 shows that an average of 14.9% of the classes of the Swing framework applications are FICs. In

terms of LOC, an average of 13.9% of the LOC of the Swing framework applications are for FICs. For applications developed using domain frameworks, much higher percentage averages were found. Table 5.3 shows that an average of 68.5% of the classes of the SalesPoint framework applications are FICs. In terms of LOC, an average of 75.5% of the LOC of the SalesPoint framework applications are for FICs.

Application Name	Number of classes	Number of LOC	Number of FICs	Number of LOC in FICs
Student management system	47	3887	31 (66%)	1568 (40.3%)
Chatting system	55	7464	3 (5.5%)	179 (2.4%)
Course management system	44	3191	17 (38.6%)	667 (20.9%)
StoneClash Strategy Game	106	5324	56 (52.8%)	2050 (38.5%)
Army Game	149	8792	66 (44.3%)	3449 (39.2%)
Average	80.2	5731.6	**41.4%**	**28.3%**

Table 5.1: Applications developed using CSF

Application Name	Number of classes	Number of LOC	Number of FICs	Number of LOC in FICs
Hook Master	112	10520	9 (8%)	611 (5.8%)
Java Master	66	3846	4 (6.1%)	251 (6.5%)
Chatting system	55	7464	21 (38.2%)	2639 (35.4%)
Course management system	44	3191	7 (15.9%)	425 (13.3%)
StoneClash Strategy Game	106	5324	23 (21.7%)	2117 (39.8%)
Army Game	149	8792	15 (10.1%)	1797 (20.4%)
Tiler shop system	39	3114	4 (10.3%)	159 (5.1%)
Photo-service system	76	8831	10 (13.2%)	493 (5.6%)
Casino system	41	8859	2 (4.9%)	69 (0.8%)
Pizza shop system	59	4516	12 (20.3%)	182 (4%)
Average	74.7	6445.7	**14.9%**	**13.7%**

Table 5.2: Applications developed using the Swing framework

Application Name	Number of classes	Number of LOC	Number of FICs	Number of LOC in FICs
Fast food shop system	18	1161	13 (72.2%)	890 (76.7%)
Tiler shop system	39	3114	28 (71.8%)	2174 (69.8%)
Photo-service system	76	8831	41 (53.9%)	6659 (75.4%)
Casino system	41	8859	25 (60.1%)	5042 (56.9%)
Golf club system	50	5041	45 (90%)	4821 (95.6%)
Pizza shop system	59	4516	37 (62.7%)	3534 (78.3%)
Average	47.2	5253.7	**68.5%**	**75.5%**

Table 5.3: Applications developed using the SalesPoint framework

5.1.4. Using multiple frameworks

When multiple frameworks are used to build an application, the number of FICs is equal to the summation of the number of FICs created using the hooks of each of the frameworks. In our case study, eight applications use two frameworks. Table 5.4 shows the application names, used frameworks, the total number of FICs, and their total LOC. The last row of the table calculates the average of the total number of FICs by summing the percentages of the corresponding columns in tables 5.1, 5.2, and 5.3 and dividing the result by the total number of summed percentages. The same calculation method is applied for the total number of LOC of the FICs in the last row of the table.

Table 5.4 shows, not surprisingly, that the average of the total number of FICs counted by considering all the frameworks used in the applications is much higher than the average obtained by considering only one of the used frameworks for each application. A similar result is found for the total number of LOC in the FICs. This means that if an application uses multiple frameworks, considering the reusable test cases of all of the used frameworks in an application can reduce the class testing time more, on average, than considering the reusable test cases of one framework only.

Application	Used frameworks	Total number of FICs	Total number of LOC in FICs
Chatting system	Swing & CSF	24 (43.6%)	2818 (37.8%)
Course management system	Swing & CSF	24 (54.5%)	1092 (34.2%)
StoneClash Strategy Game	Swing & CSF	79 (74.5%)	4167 (78.3%)
Army Game	Swing & CSF	81 (54.4%)	5246 (59.7%)
Tiler shop system	Swing and SalesPoint	32 (82.1%)	2333 (74.9%)
Photo-service system	Swing and SalesPoint	51 (67.7%)	7152 (81%)
Casino system	Swing and SalesPoint	27 (65.9%)	5111 (57.7%)
Pizza shop system	Swing and SalesPoint	49 (83.1%)	3716 (82.3%)
Average		**64.6%**	**63.2%**
Average using one framework		**32.8%**	**31.7%**

Table 5.4: Framework applications that use multiple frameworks

5.1.5. Conclusions

The case study examined the reusability of the FICs in several framework applications. The case study showed that a high percentage of the classes of applications developed using domain frameworks are FICs, while the percentage of the FICs in applications developed using application frameworks varies largely according to the specification domains of the framework and the applications. The results support the hypothesis that the reusability of the FICs in the applications developed using domain frameworks is likely to be greater than the reusability of the FICs in the applications developed using application frameworks. At the framework development stage, reusable test cases can be generated for the FICs to be used at the application development stage. As the percentage of the FICs increases in the application, the part of the application tested using the reusable test cases increases and the amount of testing work required at the application development stage reduces.

Typically, building reusable test cases is a costly task. The case study results indicate that it is worthwhile to build reusable test cases for applications developed using domain frameworks as the original investment will be recouped after producing a few number of framework applications. However, it might not be worthwhile to build reusable test cases for some application frameworks because of the relatively low percentage of the FICs in the applications developed using the frameworks. Finally, in cases involving multiple frameworks, the case study results show that considering the reusable test cases provided with all the frameworks used in an application can save more testing time than using the reusable test cases provided with one framework.

Application developers can add new specifications to the FICs at the application development stage. These specifications are not covered by the reusable test cases built at the framework development stage. This means that the reusable test cases can cover part of the implemented FICs but not all. In the next two case studies, we study the percentage of the specifications, in terms of the number of transitions in the state model, of the implemented FICs covered by the reusable test cases.

5.2. Case Study 2: Testing CSF Applications

This section evaluates the proposed test case generation and use techniques in terms of practicality and reusability of test cases using five CSF applications. First, it discusses a concrete example from one of the CSF applications. The example shows how the proposed test case generation technique can be used to generate reusable test cases for FICs of the CSF applications and shows how to use the test cases to test an implemented FIC. In addition, the case study considers five CSF applications and measures the percentage of the tested specifications of the implemented FICs using the reusable test cases. The tested specifications are measured in terms of the number of covered transitions in the state-transition testing model that specifies the FIC behavior. The results show that, on average, a relatively high percentage of the testing model transitions of the implemented FICs in the CSF applications are covered using the reusable test cases determined at the CSF development stage, which therefore reduces the testing time considerably for these applications. The introduced techniques in this thesis are applied for the CSF and its applications manually, which takes considerable time. However, the tool introduced in Chapter 6 can automate a large portion of the testing process, which reduces the time required for applying the techniques.

5.2.1. Generating reusable test drivers for CSF

The CSF hooks describe the behavior of ten FICs and show how they can be implemented or customized. However, the set of available hook descriptions does not describe how to use all the methods of the extended framework classes, which forced us to read the Javadoc document of the extended framework classes and even to go through the framework code and communicate with the framework developer to write the preconditions and postconditions of the FIC methods inherited from the framework classes and not specified in the CSF hook descriptions. Specifically, the set of available hook descriptions define 66 methods for the ten FICs out of 122. After that, we have followed the rules explained in Section 3.2 to synthesize manually the testing models for the ten FICs. The synthesized models consist of a total of 94 states and 2261 transitions. Finally, we have determined the paths required to build the test cases using the all paths-state technique. The actual test cases were not built because they are not required to

derive the required results and because it takes a considerable time, in the absence of a supporting tool, to build test cases for the complex CSF FICs using the all paths-state technique.

The most complex FIC example in the CSF is called *NewCAO*. The *NewCAO* FIC is a class defined in the CSF hooks to extend the *CommAwareObject* CSF class. The *CommAwareObject* framework class is used to create objects that can communicate across the network. It consists of seven public methods that operate on a private instance variable. The CSF hooks, which define the *NewCAO* class, also define 24 extra methods that operate on three additional private instance variables. Moreover, the hooks specify the name of the FIC (i.e., *NewCAO*), the names of the instance variables, the names of the methods, the method parameters, and the method specifications. The class state-based testing model synthesis technique was applied to build the class-testing model for the *CommAwareObject* class using the hook descriptions [CSF], as described in Section 3.2. The model consists of 50 states including alpha and omega states and 1820 transitions [NewCAO Model].

Figure 5.1 shows a test driver example generated using the all paths-state technique. The test driver checks whether the mail server of the *NewCAO* can be set as proposed by the framework hooks. The name of the test driver class *TEST16_NewCAO* consists of (1) the keyword TEST to indicate that the class is implemented for testing purposes, (2) the test driver identifier within the test suite of the *NewCAO* FIC, and (3) the name of the FIC for which the test driver is generated. The test driver class contains only a constructor method *NewCAO()* in which a state-transition model path consists of two transitions is traversed. In Line 3 of the test driver, the event associated with the first transition is implemented and, in Line 4, the state-invariants of the resulting state are checked. In Line 5, the parameter of the event associated with the second transition is set and, in Line 6, the event is implemented. Finally, in lines 7 and 8, the action associated with the second transition and the state-invariants of the resulting state are checked.

5.2.2. Testing CSF applications

In this case study, five CSF applications developed by fourth-year undergraduate students at the University of Alberta were randomly selected out of a pool of 15 applications. In an application called *Chatting System*, the application developer implemented a *NewCAO* FIC. The implemented class is named *MTA*. This section shows the implementation and testing steps of the MTA class.

```
1   public class TEST16_NewCAO{
2       public TEST16_NewCAO(){
3            NewCAO o = new NewCAO();
4            /** @assert((o.getInbox()==null)&&  (o.getOutbox==null)&&
                          (o.getMailServer()==null)&& ...) */
5            Address add=new Address("server",3029,"inbx");
6            o.setMailServer(add);
7            /** @assert(o.getMailServer().getAddress().equals(add)) */
8            /** @assert((o.getInbox()==null)&&  (o.getOutbox==null)&&
                        (o.getMailServer()!=null)&& ...) */
8       }
9   }
```

Figure 5.1: Sample test driver generated for the *NewCAO* FIC.

- Implementing the MTA class

As shown in Table 5.5, in MTA class, the application developer renamed some of the methods introduced by the hooks, removed some parameters of some methods, and reused the name of some methods as-is. The application developer did not use the rest of the methods defined in the CSF hooks in the MTA class implementation and, therefore, these methods are not included in Table 5.5. MTA class extends the *CommAwareObject* class and inherits its methods. The application developer customized some of the MTA methods built using the hooks and added two instance variables and seven methods to the MTA class. Figure 5.2 provides a partial listing of the code of the MTA class (i.e., the contents of the method blocks are not listed because of space limitations).

- Testing the MTA class

To use the test cases built at the CSF development stage in testing the MTA class, the contents of Table 5.5, the implemented MTA class, and the state-transition model created at the framework development stage for the *NewCAO* "virtual" class have to be provided. The contents of Table 5.5 are used to determine the reused transitions. We have removed

the transitions corresponding to the ignored specifications from the state-transition model of the *NewCAO* class. This results in six states and 57 transitions as shown in Table 5.6. For the transitions, the table shows only the events associated with the transitions. The predicates and actions associated with the transitions are not shown because of space limitations. Each cell in the table lists the events associated with the transitions that have the row source state and the column destination state. For example, the cell in the third row and third column shows the events associated with the twelve transitions from state *s2* to the same state. Five of the transitions, indicated with asterisks, model illegal behaviors. The other seven transitions model legal behaviors. As shown in the first column of Table 5.5, there are two *send* methods that have different signatures defined in the CSF hooks. Each of them throws an exception (i.e., illegal behavior). Therefore, there are four transitions associated with the two *send(...)* events (i.e. one for each signature), from state *s2* to the same state: two model legal behaviors and the other two model illegal behaviors.

Method declaration is CSF hooks	Method declaration in MTA class
NewCAO()	MTA()
getAddress()	getAddress()
setOutbox(Address)	setDefaultAddress(Address)
shutdown()	shutdown()
send(Address,Address, String, Data)	sendMessage(Address, Data) sendMessage(Data) sendSyncMessage(Address, Data) sendSyncMessage(Data)
send(Vector, Address, String, Data)	sendMessage(Vector, Address, String, Data)

Table 5.5: Methods defined in CSF hooks and used in MTA class

All test case identifiers associated with the removed transitions were removed from the set of test case identifiers associated with the remaining transitions. The set of the test case identifiers associated with the remaining transitions is the set of the reusable test cases. All other test cases are not applicable for testing the MTA class.

The preconditions and postconditions of the added methods extracted from the MTA class Java file were used to determine the states and transitions to be added to the provided state-transition table. For our example, four transitions were added as self loop

transitions to each of the states *s1*, *s2*, *s3*, and *s4* extracted for the *NewCAO* FIC. The events associated with the transitions are shown in the first four columns of Table 5.7. Moreover, it was found, using the model synthesis technique discussed in Section 3.2, that the states *s1*, *s2*, *s3*, and *s4* included in the provided model have to be partitioned each into four states, which created sixteen more substates. For the added states, 72 transitions were added to model the added specifications as shown in Table 5.7. For each of the superstates, the substates and transitions associated with them form a sub-state transition model. The sub-state transition models for the four superstates are identical. The table entries in the last four rows and last four columns of Table 5.7 show the sub-state transition model of any of the superstates. In the table, *s(1-4).1* denotes the first substate of any of the four superstates.

```
public class MTA extends CommAwareObject
{
   //instance variables and methods implemented using the framework hooks
   protected MailServer m_mailServer;
   protected Inbox      m_inBox;
   protected Outbox     m_outBox;
   protected SyncSend   m_sync;
   protected Address    m_defaultRecipient;
   public MTA(){ … }
   public Address getAddress() { … }
   public Address getDefaultAddress() { … }
   public void shutDown() { … }
   public void setDefaultRecipient(Address addr) { … }
   public void sendMessage(Address addr, Data msg) throws MTAException { … }
   public void sendMessage(Data msg) throws MTAException { … }
   public void sendMessage(Vector recipient,Data msg) throws MTAException {…}
   public void sendSyncMessage(Address addr,Data msg) throws MTAException {…}
   public void sendSyncMessage(Data msg) throws MTAException { … }

   // new instance variables and methods added by the application developer
   protected Dispatcher m_dispatcher;
   protected Vector     m_monitors;
   static public MTA getMTA() { … }
   public void registerHandler(ILocusHandler h) { … }
   public void unregisterHandler(ILocusHandler h) { … }
   public void registerMonitor(ILocusMonitor m) { … }
   public void unregisterMonitor(ILocusMonitor m) { … }
   public void notifyHandlers(Address from, LocusMsg msg) { … }
   public void sendFailed(SendException ex) { … }
}
```

Figure 5.2: The MTA partial code

STATE	alpha	s1	s2	s3	s4	omega
alpha	MTA()*		MTA()			
s1		setOutbox(...)*,send(...)*,send(...)*,handleMessage(...)*,messageNotify(...)*,shutdown()*,getAddress()*,messageNotify(...),finalize(),handleMessage(...),registerHandler(...)		setOutbox()		dtar
s2		shutdown()	setOutbox(...)*,send(...)*,send(...)*,handleMessage(...)*,messageNotify(...)*,getAddress(),send(...),send(...),messageNotify(...),finalize(),handleMessage(...),registerHandler(...)		setOutbox()	dtar
s3				setOutbox(...)*,send(...)*,send(...)*,handleMessage(...)*,messageNotify(...)*,shutdown()*,getAddress()*,messageNotify(...),finalize(),handleMessage(...),registerHandler(...)		dtar
s4				shutdown()	setOutbox(...)*,setOutbox(...),send(...)*,send(...)*,handleMessage(...)*,messageNotify(...)*,getAddress(),send(...),send(...),messageNotify(...),finalize(),handleMessage(...),registerHandler(...)	dtar
omega						

Table 5.6: The state-transition table of the MTA excluding the added states and transitions (* Transition that models an illegal behavior)

STATE	s1	s2	s3	s4	s(1-4).1	s(1-4).2	s(1-4).3	s(1-4).4
s1	registerHandler(…),getMTA(), sendFailed(…), notifyHandlers(…)							
s2		registerHandler(…), getMTA(),sendFailed(…),notifyHandlers(…)						
s3			registerHandler(…), getMTA(),sendFailed(…),notifyHandlers(…)					
s4				registerHandler(…), getMTA(),sendFailed(…),notifyHandlers(…)				
s(1-4).1					unregisterHandler(..)*,unregisterMonitor (…)*	registerHandler(…)	registerMonitor(…)	
s(1-4).2					UnregisterHandler (…)	registerHandler(…),unregisterHandler(…)		registerMonitor(…)
s(1-4).3					UnregisterMonitor (…)		registerMonitor(…), unregisterMonitor(…)	RegisterHandler(…)
s(1-4).4						UnregisterMonitor (…)	unregisterHandler(…)	registerHandler(…),unregisterHandler(…),registerMonitor(…),unregisterMonitor(…)

Table 5.7: The transitions that model the MTA class added specifications

(* Transition that models an illegal behavior)

To reuse the test cases built at the framework development stage, the mapping class *NewCAO* shown in Figure 5.3 is developed using Table 5.5. This class inherits the implemented class (i.e., MTA class) and maps the methods introduced by the CSF hooks to the ones used in the MTA class. As an example, when the application developer implemented the MTA class, the *setOutbox* method is renamed *setDefaultRecipient*. Therefore, when test drivers call the *setOutbox* method, the *setDefaultRecipient* method of MTA should be called. As shown is Figure 5.3, this is coded as:

public void setOutbox(Address Add) { super.setDefaultRecipient(Add); }

```
public class NewCAO extends MTA {
       public NewCAO() {
              super();
       }
       public void send(Address toAddr,Address retAddr,String tp,Data aData){
              switch ((new DRIVER_MTA).getSwitchKey()) {
                     case 1:super.sendMessage(toAddr, aData);
                            break;
                     case 2:super.sendMessage(aData);
                            break;
                     case 3:super.sendSyncMessage(toAddr, aData);
                            break;
                     case 4:super.sendSyncMessage(aData);
                            break;
          }
       }
       public void send(Vector v, Address retAddr, String tp, Data aData) {
              super.sendMessage(v,retAddr,tp,aData);
       }
       public void setOutbox(Address Add) {
              super.setDefaultRecipient(Add);
       }
       public Address getAddress() {
              return super.getAddress();
       }
       public void shutDown() {
              super.shutDown();
       }
}
```

Figure 5.3: *NewCAO* mapping class

When the MTA class was implemented, some of the parameters introduced for the *send* method were ignored. Therefore, in Figure 5.3, the implemented version of the *send* method renamed *sendMessage* in the MTA class includes:

public void send(Address toAddr,Address retAddr, String tp, Data aData) {

...

```
        case 1: super.sendMessage(toAddr, aData);
        ...
}
```

In the MTA class, the *send(Address toAddr,Address retAddr, String tp, Data aData)* method introduced in the CSF hooks has four different implementations. To test the four implementations using the reusable test cases, the following code is included in the *NewCAO* class as shown in Figure 5.3:

```
public void send(Address toAddr,Address retAddr, String tp, Data aData) {
        switch ((new DRIVER_MTA).getSwitchKey()) {
                case 1:super.sendMessage(toAddr, aData); break;
                case 2:super.sendMessage(aData); break;
                case 3:super.sendSyncMessage(toAddr, aData); break;
                case 4:super.sendSyncMessage(aData); break;
        }
}
```

Finally, a driver class for the test drivers was implemented. The driver invokes the constructor methods of the test drivers. Part of the driver class for the MTA test drivers is shown in Figure 5.4. In our example, the *NewCAO* class uses the *switchKey* global variable to determine which implementation to be exercised by the test drivers in case of calling the *send* method. For example, the test drivers that exercise the *send* method are invoked four times in the driver class as shown in Figure 5.4.

To perform the actual testing, the MTA class and the test drivers can be compiled using the Jcontract [Jcontract] compiler, which translates the DbC Javadoc comments used to check the resulting actions and state-invariants into bytecode. The *NewCAO* and *DRIVER_MTA* classes can be compiled using regular Java compiler. Finally, the *DRIVER_MTA* class can be executed and the results of the code generated from the DbC Javadoc comments can be checked at run time using the Jcontract tool, which reports the testing results.

```
public class DRIVER_MTA{
    private static int switchKey=1;
    public int getSwitchkey() {
              return switchKey;  }
    public static void main(String args[]){
         /* invoke all the applicable test drivers while the switchKey
            value is set to 1  */
         …
         /* set switchKey value to 2 and invoke the test drivers that invoke
            the send method */
         switchKey=2;
         new TEST17_NewCAO();
         …
         /* set switchKey value to 3 and invoke the test drivers that invoke
            the send method */
         switchKey=3;
         new TEST17_NewCAO();
         …
         /* set switchKey value to 4 and invoke the test drivers that invoke
            the send method */
         switchKey=4;
         new TEST17_NewCAO();
         …
    }
}
```

Figure 5.4: Part of the DRIVER_MTA class

5.2.3. Transition coverage results

The analysis applied for the MTA class was also applied for all implemented FICs in the five randomly selected applications of CSF. Table 5.8 shows the transition coverage results when the proposed techniques were applied for testing the implemented FICs in the CSF applications. The second column of the table provides the total number of classes implemented at the application development stage and does not include the number of used framework classes. The third column provides the number of implemented FICs in the applications. The implemented FICs are part of the classes implemented at the application development stage. The fourth column gives the total number of transitions in the state-transition model of the implemented FICs in the applications. The fifth column provides the total number of implemented FIC transitions covered by the reusable test cases as-is (i.e., with no augmentation).

Table 5.8 shows that, on average, a considerable part of the implemented application classes in the considered CSF applications (41.4%) is composed of FICs, which makes it worthwhile to build reusable test cases at the framework development stage. Moreover,

the table shows that, on average, a high percentage (76.9%) of the transitions in the state-transition models of the FICs are covered using the reusable test cases without modifying them. The coverage of the rest of the transitions requires augmenting some of the reusable test cases. None of the transitions requires building test cases from scratch.

Application name	Number of classes	Number of FICs	Number of transitions in FICs	Number of FIC transitions covered by the test cases as-is
Student management system	47	31 (66%)	413	369 (89.3%)
Chatting system	55	3 (5.5%)	113	68 (60.2%)
Course management system	44	17 (38.6%)	456	322 (70.6%)
StoneClash Strategy Game	106	56 (52.8%)	761	582 (76.5%)
Army Game	149	66 (44.3%)	1171	899 (76.8%)
Average	**80.2**	**41.4%**	**583**	**76.9%**

Table 5.8: The results of using the CSF reusable test cases for testing CSF applications

5.2.4. Conclusions

In this case study, a concrete example that applies the proposed techniques for the generation and use of the reusable test cases on one of the implemented FICs in a CSF application is illustrated. In addition, we have studied the coverage of the transitions in the specification models of the implemented FICs in five CSF applications using the reusable test cases. The results of the case study showed that, on average, 76.9% of the transitions in the specification models of the implemented FICs in the considered CSF applications are covered by the reusable test cases determined and used by the techniques introduced in this thesis.

5.3. Case Study 3: Testing SalesPoint Framework Applications

This section provides one more case study that shows how to deploy the proposed test case generation and use techniques in the context of the applications developed using the

SalesPoint framework. In addition, we evaluated the techniques proposed in this thesis using the analysis method applied in Section 5.2 for the CSF application. Six applications developed using the SalesPoint framework are considered.

5.3.1. Generating reusable test drivers for SalesPoint framework

First, we studied the six SalesPoint applications to find the FICs used in their development. We found that only twenty out of the 78 FICs introduced by the framework hooks were used in the considered framework applications. We followed the rules explained in Section 3.2 to synthesize the testing models for the twenty FICs. The synthesized models consist of a total of 70 states and 1552 transitions.

NewShop FIC is a class defined in the SalesPoint framework hooks to extend the *Shop* SalesPoint framework class and it has to be implemented in each framework application. The *Shop* class is responsible for central management tasks and for persistence. It consists of 44 public methods that operate on 21 instance variables. SalesPoint framework hooks, which define the *NewShop* class, describe how to use 12 of the *Shop* class methods, which forced us to determine the specifications of the other inherited methods using the framework documents. We have applied the class state-based testing model synthesis technique to build the class-testing model for the *NewShop* class using the specifications of the *NewShop* methods provided in the hook descriptions [SalesPoint] and other framework documents for missed hooks. The synthesized model consists of 5 states including alpha and omega states and 159 transitions as shown in Table 5.9.

5.3.2. Testing SalesPoint framework applications

In this case study, six SalesPoint framework applications [SalesPoint applications] developed by second year undergraduate students were randomly selected out of a pool of 22 applications. The following example shows how to use the reusable test cases to test an implementation for the *NewShop* FIC in the *FastFood System* application.

STATE	alpha	Dead	Running	Suspended	omega
alpha		NewShop()			
Dead		getCurrentUsers(),addActiveCustomer(...),removeActiveCustomer(...),onCustomerQueued(...),onCustomerUnQueued(...),onCustomerLoggedOn(...),onCustomerLoggedOff(...),addSalesPoint(...),removeSalesPoint(...),getSalesPoints(),setCurrentSalesPoint(...),setCurrentSalesPoint(...),setCurrentSalesPointIsAdjusting(),resetCurrentSalesPointIsAdjusting(),isCurrentSalesPointAdjusting(),getCurrentSalesPoint(),getShopState(),makePersistent(),makePersistent()*,makePersistent()*,restore(),restore()*,restore()*,restore()*,setObjectPersistent(...),setObjectTransient(...),getPersistentObject(...),getPersistentObjects(),setShopFrameTitle(...),setStatusFrameTitle(...),setStatusFrameVisible(...),isStatusFrameVisible(),getTimer(),setTimer(...),log(...),log(...)*,addStock(...),addStock(...)*,removeStock(...),getStock(...),addCatalog(...),addCatalog(...)*,removeCatalog(...),getCatalog(...),getTheShop(),setTheShop(...),suspend()*,resume()*shutdown()*,start()	runProcess(...),runBackgroundProcess(...)		quit()
Running		shutdown(...)	getCurrentUsers(),addActiveCustomer(...),removeActiveCustomer(...),onCustomerQueued(...),onCustomerUnQueued(...),onCustomerLoggedOn(...),onCustomerLoggedOff(...),addSalesPoint(...),removeSalesPoint(...),getSalesPoints(),setCurrentSalesPoint(...),setCurrentSalesPoint(...),setCurrentSalesPointIsAdjusting(),resetCurrentSalesPointIsAdjusting(),isCurrentSalesPointAdjusting(),getCurrentSalesPoint(),getShopState(),makePersistent(),makePersistent()*,makePersistent()*,restore(),restore()*,restore()*,restore()*,setObjectPersistent(...),setObjectTransient(...),getPersistentObject(...),getPersistentObjects(),setShopFrameTitle(...),setStatusFrameTitle(...),setStatusFrameVisible(...),	suspend()	

Table 5.9 (Part 1): The NewShop FIC state-transition table
(* Transition that models an illegal behavior)

STATE	alpha	Dead	Running	Suspended	omega
			isStatusFrameVisible(),getTimer(),setTimer(...),log(...),log(...)*,addStock(...),addStock(...)*,removeStock(...),getStock(...),addCatalog(...),addCatalog(...)*,removeCatalog(...),getCatalog(...),getTheShop(),setTheShop(...),runProcess(...)*,runBackgroundProcess(...)*,resume()*,start()*,quit()*,shutdown()*		
Suspended			resume()	getCurrentUsers(),addActiveCustomer(...),removeActiveCustomer(...),onCustomerQueued(...),onCustomerUnQueued(...),onCustomerLoggedOn(...),onCustomerLoggedOff(...),addSalesPoint(...),removeSalesPoint(...),getSalesPoints(),setCurrentSalesPoint(...),setCurrentSalesPoint(...),setCurrentSalesPointIsAdjusting(),resetCurrentSalesPointIsAdjusting(),isCurrentSalesPointAdjusting(),getCurrentSalesPoint(),getShopState(),makePersistent(),makePersistent()*,makePersistent()*,restore(),restore()*,restore()*,restore()*,setObjectPersistent(...),setObjectTransient(...),getPersistentObject(...),getPersistentObjects(),setShopFrameTitle(...),setStatusFrameTitle(...),setStatusFrameVisible(...),isStatusFrameVisible(),getTimer(),setTimer(...),log(...),log(...)*,addStock(...),addStock(...)*,removeStock(...),getStock(...),addCatalog(...),addCatalog(...)*,removeCatalog(...),getCatalog(...),getTheShop(),setTheShop(...),runProcess(...)*,runBackgroundProcess(...)*,suspend()*,start()*,quit()*,shutdown()*	
omega					

Table 5.9 (Part 2): The NewShop FIC state-transition table

(* Transition that models an illegal behavior)

- Implementing the *FastFood* class

The application developer of the *FastFood System* application created an implementation for the *NewShop* FIC and named it the *FastFood* class. As shown in Figure 5.5, the class consists of three methods defined in the SalesPoint framework hooks. Table 5.10 shows the mapping between the names of the implemented methods in the *FastFood* class and the names of the methods introduced by the hooks. The state-transition model of the *FastFood* class has the same states and transitions as the *NewShop* FIC.

```
public class FastFood extends Shop {
    public FastFood() { … }
    public MenuSheet createShopMenuSheet() { … }
    public void quit() { … }
}
```

Figure 5.5: The *FastFood* class partial code

Method declaration in SalesPoint framework hooks	**Method declaration in the *FastFood* class**
NewShop()	FastFood()
createShopMenuSheet()	createShopMenuSheet()
quit()	quit()

Table 5.10: The method-name-mapping table for the *FastFood* class

- Testing the *FastFood* class

To test the *FastFood* class, the contents of the method-name-mapping table given in Table 5.10, the implemented *FastFood* class given partially in Figure 5.5, and the state-transition model created at the framework development stage for the *NewShop* "virtual" class shown in Table 5.9 are provided. The framework hooks do not introduce any new methods for the *NewShop* FIC but the hooks show how to use some of the inherited methods of the *Shop* framework class. Therefore, the *FastFood* class inherits all the methods that determine the *NewShop* FIC (i.e., the *FastFood* class has the same states and transitions shown in Table 5.9 except for the transition associated with the invocation of the constructor method). All the reusable test cases for the *NewShop* FIC are applicable for testing the *FastFood* class. All that is needed to use the test cases is a driver class to

invoke them and the mapping class shown in Figure 5.6 to map the names of the methods used in the *FastFood* class to the ones used in the *NewShop* class and the test drivers.

```
public class NewShop extends FastFood {
        public NewShop() {
                super();
        }
        public MenuSheet createShopMenuSheet() {
                return super.createShopMenuSheet();
        }
        public void quit() {
                super.quit();
        }
}
```

Figure 5.6: *NewShop* mapping class

5.3.3. Transition coverage results

A similar table to the one shown in Table 5.8 is drawn for the six applications developed using the SalesPoint framework and considered in this case study. The resulting table is shown in Table 5.11.

Application name	Number of classes	Number of FICs	Number of transitions in FICs	Number of FIC transitions covered by the test drivers as-is
FastFood shop system	18	13 (72.2%)	1147	1135 (99%)
Tiler shop system	39	28 (71.8%)	2605	2434 (93.4%)
Photo-service system	76	41 (53.9%)	3465	3303 (95.3%)
Casino system	41	25 (60.1%)	3537	3337 (94.3%)
Golf club system	50	45 (90%)	3994	3910 (97.9%)
Pizza shop system	59	37 (62.7%)	3408	3264 (95.8%)
Average	**47.2**	**68.5%**	**3026**	**96%**

Table 5.11: The results of using the SalesPoint framework reusable test cases for testing the SalesPoint framework applications

Table 5.11 shows that, on average, a considerable portion of the implemented application classes (68.5%) is composed of FICs, which makes it worthwhile to build reusable test cases at the framework development stage. Moreover, the table shows that,

on average, a very high percentage (96%) of the transitions in the state-transition models of the FICs are covered using the reusable test cases without modifying them. The coverage of the rest of the transitions (average of 4%) requires augmenting some of the reusable test cases. None of the transitions requires building test cases from scratch.

5.3.4. Conclusions

In this case study, a concrete example that applies the proposed techniques for the generation and use of the reusable test cases on one of the implemented FICs in a SalesPoint framework application is illustrated. In addition, we have studied the coverage of the transitions in the specification models of the implemented FICs in six SalesPoint framework applications using the reusable test cases. The results of the case study showed that, on average, 96% of the transitions in the specification models of the implemented FICs in the considered SalesPoint framework applications are covered by the reusable test cases determined and used by the techniques introduced in this thesis.

On average, the percentage of the specifications of the FICs, in terms of the number of transitions in the state-transition models, covered using the reusable test cases without modifying them is greater in the SalesPoint framework applications than in the CSF applications. Given the relative numbers, we believe that this is because the amount of functionality already defined for the FICs of the SalesPoint framework by a combination of the inherited framework classes and the hooks is larger than the amount of functionality defined for the CSF FICs. One way to measure the amount of functionality is by calculating the number of methods per FIC. For the FICs of the SalesPoint framework considered in the case study, the total number of methods defined for the twenty FICs is 570 (i.e., an average of 28.5 methods/FIC), while the total number of methods defined for the ten FICs defined in the CSF hooks is 122 (i.e., an average of 12.2 methods/FIC). This gives an indication that, in general, as the amount of functionality of the FICs defined in the inherited framework classes or by the hooks increases, the amount of functionality added by the application developer to the FICs decreases. Consequently, as the amount of functionality of the FICs defined in the inherited framework class or by

the hooks increases, the portion of the FICs tested by the reusable test cases as-is at the application development stage increases.

5.4. Case Study 4: All Paths-State Coverage

In the fourth case study, we evaluate experimentally the all paths-state technique in comparison to the round-trip path and the all-transitions techniques. The comparison is performed in terms of the number of transitions covered in the updated state model after deleting transitions. Two WaveFront Pattern frameworks derived using CO_2P_3S parallel programming system [McDonald+ 97] are considered in this study. The hooks of each framework identify two FICs and their specifications. One of the FICs is very simple (i.e., trivial) and, therefore, it is not considered in this study. The characteristics of the statechart of the other one, for each framework, are shown in the first row of Table 5.12.

	FIC of Framework 1	**FIC of Framework 2**
Statechart		
No. of states	7	7
No. of transitions	37	35
Round-trip path tree		
No. of nodes	38	36
No. of edges	37	35
No. of test cases	33	33
All transitions		
No. of test cases	37	35
All paths-state tree		
No. of nodes	116	151
No. of edges	115	150
No. of test cases	95	124

Table 5.12: High level descriptions of the used graphs

5.4.1. Case study settings and results

The case study considered three test case coverage techniques: round-trip path, all transitions, and all paths-state. The trees corresponding to the round-trip path technique and all paths-state technique are constructed from the statechart of the considered FICs. For each of the two considered statecharts, there are four possible different round-trip

path trees that can be constructed from them using the round-trip path tree production procedure [Binder 99]. However, they all have the same number of states, nodes, and number of generated test cases and these numbers are shown in Table 5.12. The all-transitions technique generates a test case for each outgoing transition from a state in the statechart. Since some states can be reached using different paths, we have to select one path and write the corresponding test case. In our study, we followed the algorithm provided in [Offut+ 99] to find a path to a state. If there is more than one path to a state, the algorithm picks one of the paths. The selection of the path affects greatly the results of the case study. Therefore, in our analysis we considered each path alone, obtained the required results, and computed the average over all of the considered paths. For the transitions of the FIC in the first and second framework, 136 and 140 paths were considered, respectively. Finally, there is only one possible all paths-state tree for each statechart and its characteristics are shown in the last row of Table 5.12.

The study considered an application for each framework. The applications were not built for the purpose of the study. Statecharts were drawn for the implemented FICs in the applications. The names of the implemented FICs in the first and second applications are *SkylineMatrixInterface* and *MatrixBlock*, respectively. For each of the two implemented FICs, three transitions of the original statecharts were removed because they are not required in modeling the specifications of the implemented FICs. This results in having 34 and 32 reused transitions in the statecharts that specify the behaviors of the *SkylineMatrixInterface* and *MatrixBlock* objects, respectively. The effect of the removed transitions on the baseline test cases was analyzed for each of the three testing techniques. In the analysis, all broken baseline test cases were discarded. Finally, we counted the number of statechart transitions of the *SkylineMatrixInterface* and *MatrixBlock* objects that are still covered using the non-broken test cases. Since there are four different possible round-trip path trees for each of the statecharts, we have considered each round-trip path tree alone and, then, we have computed the average number of transitions covered by the non-broken test cases. We did the same analysis for the all-transitions test cases, because there are different possible paths to each state.

Figure 5.7 shows the average number of the transitions of the implemented FICs covered by the non-broken test cases when each of the three techniques is used to produce test cases for *SkylineMatrixInterface* and *MatrixBlock* classes. For example, the all paths-state coverage technique was used to generate test cases for the considered FIC of Framework 1. Table 5.12 shows that 95 test cases were required. The implementation of the FIC, i.e., *SkylineMatrixInterface* class, reused 34 out of 37 transitions introduced by the hooks. The test cases that cover the non-reused transitions (i.e., the transitions not required in modeling the specifications of the implemented FICs) were discarded because they cannot be used as-is. Figure 5.7 shows that the remaining test cases were able to cover 34 reused transitions (i.e., all the reused transitions).

The results showed that the all paths-state technique produces test cases that are more effective than the ones produced using the all-transitions and round-trip path techniques in covering the reused transitions in the specification models of the implemented FICs.

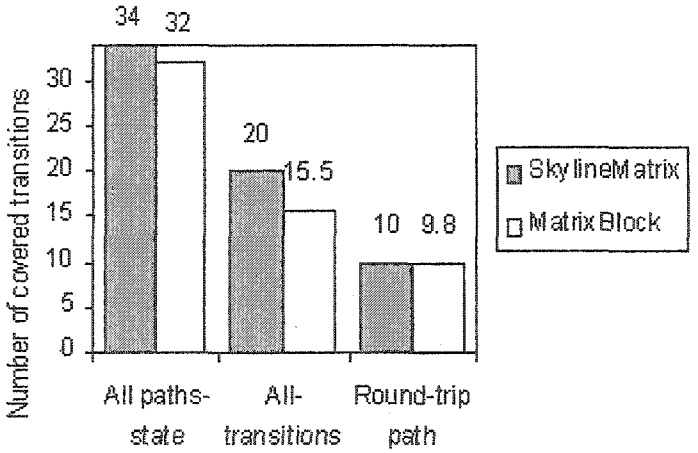

Figure 5.7: Coverage comparison results

5.4.2. Conclusions

The main conclusion of this final case study is that the all paths-state technique produces test cases that cover more transitions in the specification models of the implemented FICs than the test cases generated using the round-trip path or all-transitions techniques. In the case study, the test cases built using the all paths-state technique were able to cover all the reused transitions in the applications. This supports experimentally

Property 3.2 which states that when a transition in the state model is deleted, the non-broken test cases built using the all paths-state technique cover all remaining transitions in the state model initiated from the reachable states.

The degree of coverage for the transitions in the specification models of the implemented FICs achieved using techniques other than the all paths-state technique depends greatly on the topology of the state-transition model (i.e., the way in which the model is connected). For example, for a model that has test cases built using the round-trip path coverage technique, after deleting a transition from the model, all the transitions of the updated model are guaranteed to be covered by the non-broken test cases if there is only one path in the model from the alpha state to the destination state of the deleted transition. Otherwise, the non-broken test cases may not cover all the reused transitions.

Performing the analysis done in this case study for the FICs that have a relatively large number of states and transitions requires an exhaustive effort. In this case, a large number of possible round-trip path trees have be constructed to perform the analysis for the round-trip path technique and a huge number of possible paths to cover the transitions have to be considered to perform the analysis for the all-transitions technique. Therefore, although we did not repeat the case study for the FICs of other frameworks, we can conclude from this case study that as the number of specifications that are ignored (i.e., not used) in application development increases, the same or better is the relative performance of the all paths-state coverage technique.

5.5. Summary

This chapter studied the relationship between the type of the framework and the use of the FICs in the framework applications. Moreover, it studied the specification size, in terms of the number of transitions in the state-transition models, of the FICs typically covered by the reusable test cases at the application development stage. Finally, the all paths-state technique was compared to other coverage techniques in terms of the coverage sensitivity of the test cases due to a transition deletion. Fifteen applications

developed using four frameworks were used in the case studies. The main conclusions are:

1. In our case studies, the reusability of the FICs in the applications developed using domain frameworks is greater than the reusability of the FICs in the applications developed using application frameworks. Specifically, on average, 68.5% of the classes in the considered applications developed using SalesPoint, a domain-oriented framework, are FICs, while, on average, 14.9% and 41.4% of the classes in the considered applications developed using the application-oriented frameworks Swing and CSF, respectively, are FICs.

2. In the case studies, a high percentage (i.e., an average of 76.9% for CSF applications and 96% for SalesPoint applications) of the transitions in the state-transition models of the implemented FICs are tested using reusable test cases built without modifications at the framework development stage. Therefore, a considerable class-testing cost is saved at the application development stage when the reusable test cases generated at the framework development stage are used.

3. In the case studies, the all paths-state technique produces test cases that cover number of transitions in the specification models of the implemented FICs the same as or more than the number of transitions covered by the test cases generated using the round-trip path or all-transitions techniques. The relative transition coverage of the all paths-state technique remains the same or increases as the number of specifications that are ignored in application development increases.

Chapter 6 Tool Support

6.1. Introduction

Automation is a vital issue in software testing. Typically, many test cases have to be built and evaluated, which makes manual testing impractical for medium to large sized software systems. In this thesis, we have seen how FICs introduced by the framework hooks are the reusable classes for which reusable test cases can be built and applied. This chapter addresses the automation issues related to the generation and use of the FIC reusable test cases. For this purpose, a tool called *Framework Interface State Transition Tester ($FIST_2$)* is introduced and its prototype is developed. The tool generates reusable test cases for Java framework FICs at the framework development stage. The tool also deploys, executes, and evaluates the test cases at the application development stage.

6.2. The $FIST_2$ Tool

$FIST_2$ is a tool that supports testing the implemented FICs at the class-testing level using reusable test cases. The tool supports the testing at two main stages: (1) framework development stage and (2) application development stage. At the framework development stage, the framework developer builds the framework, documents it, and uses the $FIST_2$ tool to generate class state-based reusable test cases for the FICs. At the application development stage, the application developer builds the application and uses the $FIST_2$ tool to assist in building and testing the implemented FICs. In the following discussion, all the $FIST_2$ tool components and specifications are implemented in the prototype version unless otherwise specified. Appendix D shows a complete example that uses the tool at the framework development stage to generate reusable test cases for the *NewAccount* FIC example and then reuses the test cases at the application development stage to test an implemented version of the FIC.

6.2.1. Using $FIST_2$ at the framework development stage

At the framework development stage, the $FIST_2$ tool supports the construction of the state-transition models of the FICs and the generation of the reusable test drivers using the constructed models.

- Tool Inputs

The $FIST_2$ tool requires several inputs at the framework development stage as follows:

1. *Framework hooks*. Framework hooks define the specifications (i.e., preconditions and postconditions) of the FIC methods introduced by the hooks. These method specifications are used to synthesize the state-based testing model of the FIC at the framework development stage. In addition, they are used as test oracles at the application development stage. In the prototype version of the tool, the method specifications are not extracted from the hooks automatically because the module responsible for constructing the state-transition model from the method specifications is not yet implemented. Instead, the user has to construct the model manually from the method specifications listed in the hooks using the algorithms illustrated in Section 3.2. The user is, however, provided with a friendly GUI to input the model description in a tabular form.

2. *Non-event-driven transitions*. Non-event-driven transitions cannot be synthesized automatically using the algorithms illustrated in Section 3.2. The user of the tool has to determine the source and destination states of the non-event-driven transitions. The tool automatically produces the predicates of the transitions.

3. *Predicate implementation*. Recall from Section 3.2, transitions of the FIC synthesized testing model can be associated with predicates that have to be satisfied to execute the transitions. The predicates can be as simple as a variable definition or they can involve defining a large data structure for which it is difficult to generate code to satisfy the predicate. The user of the tool has to provide the code required to satisfy the predicates of the transitions at the framework development stage. Writing the pieces of code that implement complex predicates

can be a costly task; however, this cost cannot be avoided in any state-based testing technique. The good news is that the implementation of the predicates is provided just once at the framework development stage and reused each time an application is developed at the application development stage. In most situations, the original investment can be recouped after producing a few framework applications.

- Tool Outputs

At the framework development stage, the $FIST_2$ tool has several outputs. These outputs are used later at the application development stage to test the framework applications. The outputs are as follows.

1. *Class state-based testing model.* The $FIST_2$ tool synthesizes the class state-based testing models of the FICs at the framework development stage. As mentioned earlier, the prototype version of the tool interacts with the user to specify the states and transitions of the model in a tabular form. The tool translates the tabular form of the model into a text using a special purpose language, TSTMD (Testable State-transition Model Description), described in Appendix C.

2. *Model checking report.* The $FIST_2$ tool checks that the class state-based testing model has one entry and one exit state and each state can be reached from the entry state. It then reports the checking results.

3. *FIC test drivers.* The $FIST_2$ tool uses the class state-based testing models of the FICs to generate test drivers using the all paths-state coverage technique. The test drivers are executed later at the application development stage to test the implemented FICs in the framework applications.

4. *Stubs.* The $FIST_2$ tool analyzes the hook descriptions and uses the information provided in the changes section of the hook description to determine and generate the stubs. These stubs are required at the application testing stage to isolate the

FICs. The developed prototype version of the tool does not produce the stubs; instead, the user of the tool has to provide the stubs.

- Tool Components

Several tool components are used at the framework development stage as follows.

1. *FIC state-transition table builder.* The FIC state-transition table builder component uses the method specifications of the FICs to synthesize the state-transition models. The models are provided to the user in a tabular form to be updated. As indicated earlier, this component is not implemented in the prototype version of the tool. Instead, a component that translates the tabular form of the model into a text using the TSTMD language is implemented.

2. *Model checker.* The model checker component checks the correctness of the FIC model. In the developed prototype version of the tool, the model checker component checks that there is only one entry and one exit state in the model. In addition, the component checks that all the model states are reachable from the entry state. The component can be further extended to check other state-based model properties [Binder 99] such as each guard condition is mutually exclusive of all other guards for a transition and the evaluation of the guard expression does not produce any side effects in the class under test.

3. *All paths-state test driver builder.* The all paths-state test drivers builder component applies the all paths-state coverage technique to produce the test drivers for the FICs. In addition, it uses the hook descriptions to determine and generate stubs required at the application testing stage to isolate the FICs. This latter function is not implemented in the prototype version of the tool.

- Testing Process

Figure 6.1 shows the high-level design of the tool when used at the framework development stage. The user (typically the framework developer in a test case generation role) selects the framework. The framework is stored in a database that contains the

framework code and the descriptions of the hooks. The tool passes the hook descriptions to the *FIC state-transition table builder* module. The FIC state-transition table builder module parses the preconditions and postconditions of the FIC methods, analyzes them, and produces the state-transition table for the FIC. The framework developer can edit the generated table to add the code required to satisfy the predicates of the transitions and to add the non-event-driven transitions. In the prototype version of the tool, the user develops the state-transition model manually and then interacts with the tool to create the table that describes the state-transition model. The tool translates the tabular form of the state-transition model into a text using the TSTMD language and stores the text in a file in the framework database. The user can use the *Model Checker module* of the $FIST_2$ tool to check the correctness of the model.

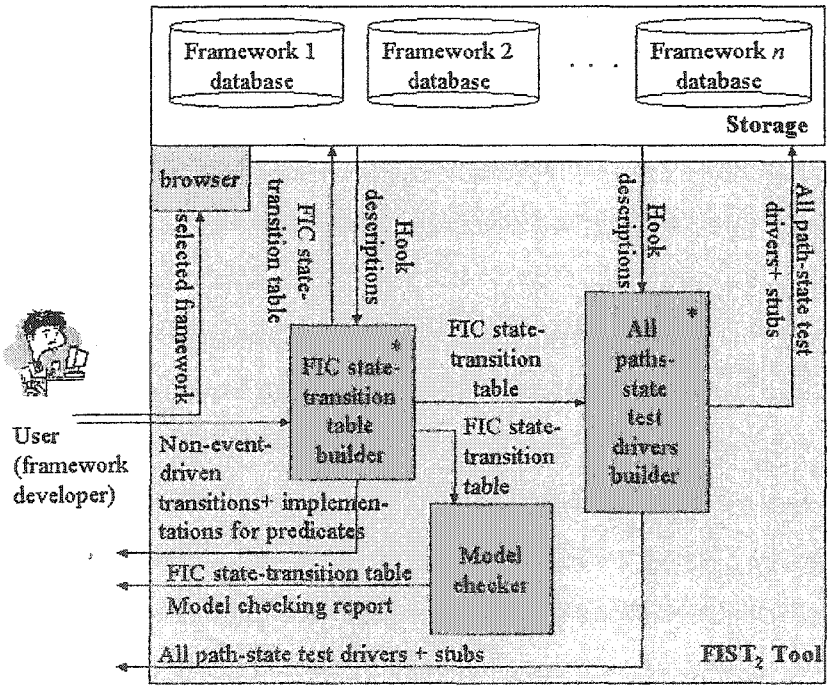

Figure 6.1: The high-level design of the $FIST_2$ tool (framework development stage)

The *All paths-state test drivers builder* component of the $FIST_2$ tool uses the state-transition table to generate the all paths-state test drivers and associates the test driver identifiers with the model transitions. In addition, it uses the hook descriptions to determine and generate the stubs required at the application testing stage to isolate the

FICs. The test drivers and stubs are stored in the framework database and provided to the user. The prototype version of the tool does not generate stubs.

6.2.2. Using $FIST_2$ at the application development stage

At the application development stage, the $FIST_2$ tool supports the use of the reusable test drivers generated at the framework development stage for Java framework FICs. The tool interacts with the Hook Master tool to construct the updated state-transition tables for the FICs, checks the correctness of the tables, determines the reusable test drivers, augments some reusable test drivers, and generates new test drivers to test new specifications. It then executes the test drivers and evaluates their results. In the prototype version of the tool, the integration between the $FIST_2$ tool and the Hook Master tool is not implemented. The other tool functions are implemented.

- Tool Inputs

The $FIST_2$ tool requires several inputs at the application development stage as follows:

1. *FIC state-transition model.* The FIC state-transition model stored in a text form at the framework development stage is used at the application development stage to model the specification of the implemented FIC.

2. *Implemented FICs.* Implemented FICs are the classes to be tested. These classes are semi-automated using the Hook Master tool. In addition, the code is instrumented by the method specifications written in the DbC language [Meyer 92] to be used as testing oracles.

3. *FIC reusable test drivers and stubs.* The FIC reusable test drivers and stubs generated and provided at the framework development stage are used at the application development stage to test the implemented FICs.

4. *Method-name-mapping table.* The method-name-mapping table is generated at the application development stage using the Hook Master tool. The table is used to generate the mapping class. The integration between the $FIST_2$ tool and the Hook

Master tool is not implemented in the prototype version of the tool and, therefore, the use of the method-name-mapping table in generating the mapping class is not implemented in the prototype version.

- Tool Outputs

At the application development stage, the $FIST_2$ tool has several outputs.

1. *Test drivers and stubs for the implemented FICs*. The $FIST_2$ tool determines the reusable test drivers and stubs, augments test drivers as necessary, and generates test drivers and stubs to test new code as necessary. The prototype version of the tool does not determine the reusable stubs or create new stubs.

2. *Driver class*. The $FIST_2$ tool generates a driver class to invoke the applicable test drivers of the implemented FIC.

3. *Class state-transition model of the implemented FIC*. The $FIST_2$ tool uses the specifications of the added methods to update the class state-transition model synthesized at the framework development stage. This output is not produced automatically in the prototype version of the tool. Instead, the user of the tool updates the tabular representation of the model according to the specifications of the implemented FIC.

4. *FIC mapping class*. The $FIST_2$ tool generates the FIC mapping class using the method-name-mapping table to map the methods defined in the hooks and used in the generated test drivers to the ones implemented in the application under test. In the prototype version of the tool, the mapping class is semi-automated. For the mapping class, the tool generates the constructor methods and the methods called in the reused transition events. The use of the method-name-mapping table in generating the mapping class is not implemented in the prototype version of the tool. Therefore, the methods of the generated FIC mapping class invoke the methods of the implemented FIC assuming that no methods are renamed. The user

of the tool has to update the names of the invoked methods according to the method-name-mapping table.

5. *Testing results.* The $FIST_2$ tool executes the test drivers and uses the Jcontract tool [Jcontract] to evaluate the testing results.

The test drivers, stubs, driver class, FIC mapping class, and the implemented FIC under test are called the testing package.

- Tool Components

Several tool components are used at the application development stage as follows.

1. *FIC state-transition table updater.* The FIC state-transition table updater component uses the specifications of the added methods to update the FIC state-transition table generated at the framework development stage. This component is not implemented in the prototype version of the tool. Instead, the user modifies the FIC state-transition table manually and then interacts with the tool to input the modifications of the FIC state-transition table.

2. *Model checker.* The model checker component is the same one used in the framework development stage.

3. *Application test drivers builder.* The application test driver builder component determines the reusable test drivers, augments test drivers as necessary, and generates new test drivers as necessary. In addition, it generates a driver class for the test drivers and uses the method-name-mapping table generated by Hook Master to generate the FIC mapping class. Finally, the application test drivers builder component produces necessary stubs. The use of the method-name-mapping table and the generation of the necessary stubs are not implemented in the prototype version of the tool.

4. *Test driver executer*. The test driver executer component compiles the classes of the testing package, executes them, and uses the Jcontract tool to evaluate the testing results.

- Testing Process

Figure 6.2 shows the high-level design of the tool when used at the application development stage. The tester selects the framework stored in a database that contains the framework code, the FIC state-transition tables, and the reusable test drivers. The user uses Hook Master to semi-automate the implementation of the FICs. Hook Master comments the Java code of the hook methods with the corresponding preconditions and postconditions specified in the hook description. The preconditions and postconditions are written in the DbC language. The user can add new code and specifications in DbC to the Java code to complete the implementation of the FIC. Hook Master also produces the method-name-mapping table that maps the methods defined in the hooks to the ones implemented in the FIC.

The $FIST_2$ tool gets from Hook Master the used FIC methods and the new methods to update the FIC state-transition tables using the *FIC state-transition table updater* module. This function is not implemented in the prototype version of the tool. Instead, the user modifies the FIC state-transition table manually and then interacts with the tool to input the modifications of the FIC state-transition table. The user can use the *Model Checker* module of the $FIST_2$ tool to check the correctness of the table. The tool stores the updated table and passes it with the reusable test drivers to the *Application test drivers builder* module, which detects the broken test drivers, augments some reusable test drivers, and generates new ones to test the new specifications not covered in the augmented test drivers. In addition, the *Application test drivers builder* module generates a driver class for the test drivers and uses the method-name-mapping table generated by Hook Master to generate the FIC mapping class. The *Application test drivers builder* module also produces the necessary stubs. The generated classes and test drivers are stored in the application database. The use of the method-name-mapping table and the generation of the necessary stubs are not implemented in the prototype version of the tool.

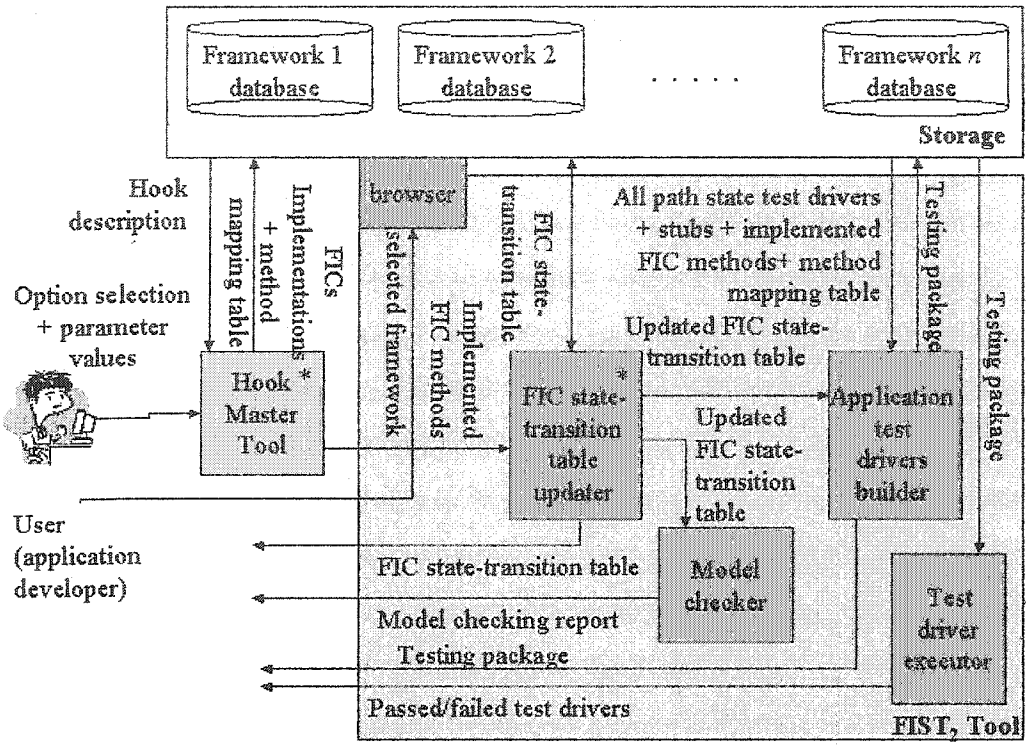

Figure 6.2: The high-level design of the FIST tool (application development stage)

The *Test driver executer* module of the $FIST_2$ tool compiles the test drivers and the implemented FICs using the dbc_javac compiler of the Jcontract tool. The Jcontract compiler checks the DbC specifications in the Javadoc comments, generates instrumented .java files with extra code to check the contracts (i.e., preconditions and postconditions) in the Javadoc comments, and compiles the instrumented .java files with the `javac` compiler. The resulting .class files are instrumented with extra bytecodes to check the contracts at runtime. Other classes, such as the mapping class and the driver class, are compiled using the regular Java compiler. Finally, the $FIST_2$ tool executes the test drivers and uses *Jcontract* tool to check automatically the contracts at runtime and report any violations found.

6.3. Summary

$FIST_2$ is a tool that supports the generation of the reusable test drivers for Java FICs at the framework development stage. In addition, it supports the use, the execution, and the evaluation of the test drivers at the application development stage. The generation of the test drivers at the framework development stage is semi-automated. The framework hooks have to be provided and the user has to provide the code required to satisfy the predicates of the state model transitions. The generation of the test drivers is performed in two main steps: (1) the construction of the class state-based testing models and (2) the use of the all paths-state coverage technique in generating the test drivers.

Once the application developer implements the FICs, typically, the use of the test drivers at the application development stage is fully automated. At this stage, the tool determines the non-broken test drivers, augments test drivers as necessary, generates new test drivers as necessary, generates mapping classes, generates stubs, and generates driver classes. Finally, the tool executes the test drivers and uses the Jcontract tool to evaluate the testing results.

At the framework development stage, the prototype version of the tool automates the generation of the reusable test cases from the state-transition model. At the application development stage, the prototype version of the tool automates the detection of the non-broken test cases, the augmentation of the test cases, the generation of the new test cases, the execution of the test cases, and the evaluation of the test cases. However, the prototype version of the tool does not build the state-transition model of the FIC automatically. Instead it interacts with the user to describe the model provided by the user. In addition, the tool does not produce stubs or interact with the Hook Master tool. The implementation of these functions is left for future work, which is discussed in the next chapter. Referring to Figures 6.1 and 6.2, the processes which are not fully implemented have an "*" in their upper right corner.

Chapter 7 Contributions and Future Work

7.1. Contributions

This research makes the following contributions in the area of OO-framework application testing.

7.1.1. Adding a new value to the hooks

Hooks were originally introduced as an aid to show where and how to extend object-oriented frameworks in constructing complete software applications. This thesis shows that the hooks can be used also as an aid to build reusable class-based test cases for the FICs implemented in the framework applications. The hooks introduce the specifications of the FIC methods. These specifications can be used by specification based testing techniques to generate test cases to test the methods as implemented in the JTest tool [Jtest]. This thesis introduces a technique to construct the state-based class testing models for the FICs using the method specifications introduced by the hooks. The method specifications are used also as test oracles to evaluate the class-based test cases. Finally, the hook descriptions identify the collaborations among the FICs and between the FICs and the framework classes. This collaboration knowledge is useful at the cluster-testing level of the framework applications. The use of the hooks in building cluster-based test cases for the framework application is an interesting additional value for the hooks, left for future work.

7.1.2. Introducing a state-based class testing model synthesis technique

The thesis introduced a new use for the method specifications. Method specifications are used in the literature as testing oracles to evaluate the results of the test cases. In addition, they are used as inputs for the process of generating method-based test cases. This thesis introduced a technique to use the method specifications in constructing state-based class testing models. These models are used to generate class-level test cases. Therefore, this technique reduces considerably the class testing cost and the chance of model construction errors and provides a consistent state-based testing model with respect to the specifications of the class methods.

7.1.3. Developing a new coverage technique

This thesis also introduces a new coverage technique to overcome the weaknesses of existing testing techniques in building reusable test cases for the FICs. Existing testing techniques, such as all-transitions, transition-pair, full predicate, and round-trip path, have weaknesses in building reusable test cases for the specifications of the FICs. When these testing techniques are used to build reusable test cases for the FICs at the framework development stage, some transitions may not have reusable test cases to cover them at the application testing stage because of some ignored specifications. The introduced technique solves the problem and, therefore, increases the degree of coverage of the test cases for the transitions of the models that represent the implemented FICs.

7.1.4. Studying the relation between the framework type and the reusability of the FICs

In this thesis, the relationship between the framework type and the reusability of the FICs is studied experimentally using three frameworks and fifteen applications. The reusability of the FICs is measured in terms of the percentage of the number of implemented FICs in the applications to the total number of classes implemented at the application development stage. The case study showed that a high percentage (average of 68.5%) of the classes of the applications developed using domain frameworks are FICs, while the percentage of the FICs in the applications developed using application frameworks varies largely (4.9-66%) according to the specification domains of the frameworks and the applications. In general, it was found that the reusability of the FICs in applications developed using domain frameworks is greater than the reusability of the FICs in the applications developed using application frameworks. As the percentage of the FICs increases in the application, the part of the application tested using the reusable test cases increases and, consequently, the amount of testing work required at the application development stage decreases.

7.1.5. Studying the reusability of the test cases developed at the framework development stage

This thesis focuses on the generation of the class-based test cases for FICs implemented at the application development stage. The test cases are generated using state-transition models. Case studies were conducted to study experimentally the percentage of the covered specifications of the implemented FICs using the reusable test cases. The covered specifications are measured in terms of the number of covered transitions in the state-transition testing model that specifies the FIC behavior. The results of the case studies show that, on average, a high percentage (average of 87.3%) of the specifications of the implemented FICs in the framework applications are tested using the reusable test drivers generated at the framework development stage, which reduces the application testing time considerably.

7.1.6. Speeding up framework application development

Frameworks provide reusable design and code which decreases the application development cost considerably. Since software testing is a time consuming and labor-intensive process, providing the framework with reusable test cases to test parts of the applications makes frameworks more appealing and encourages application developers to use the frameworks. Therefore, one of the main contributions of this thesis is in speeding up framework application development. In the thesis, we have measured the saved time indirectly by calculating the percentage of the number of FICs in the framework applications and the percentage of the number of transitions of the FIC models covered by the reusable test cases. For the considered CSF and SalesPoint framework applications, on average, the percentages of the number of FICs are 41.4% and 68.5%, respectively, and the percentages of the number of transitions in the FIC models are 76.9% and 96%, respectively.

7.1.7. Development of a supporting tool

Automation is a vital issue in software testing. Typically, many test cases have to be built and evaluated, which makes manual testing impractical. Therefore, this thesis introduces and develops a supporting tool that semi-automates the generation of the

reusable test cases at the framework developing stage. In addition, the tool automates the use of the test cases and builds test cases to test new application functionalities at the application development stage. The tool focuses on testing the FICs at the class-testing level.

7.2. Future Work

A number of interesting problems related to our research have been identified during the course of this thesis. A summary of the problems is as follows.

7.2.1. Modeling concurrent class behaviors

The proposed technique of generating the class behavior models using the method specifications is limited to classes that have sequential behaviors. Classes that have concurrent behaviors cannot be modeled using a finite state machine (FSM) model because it does not have the capability to express concurrent behaviors. In [Al-Dallal+ 97], the FSM model is extended to express concurrent behaviors. The extended FSM, UML statecharts, and Petri Nets are examples of models that can be used to express the concurrent behaviors of classes. Behavior contracts, which include method specifications, are also limited to sequential behaviors and cannot be used to express concurrent behaviors. Contracts used to express concurrent behaviors are called synchronization contracts [Beugnard+ 99]. Further research is required to show how to synthesize the testing model used to express concurrent behaviors of a class from the synchronization contracts of the class methods.

7.2.2. Evolving reusable test cases

In our research, reusable test cases are generated at the framework development stage using the hook descriptions. As the framework evolves, the hook descriptions may also evolve. Hook evolution includes adding new hook descriptions to introduce new FICs or adding new functionalities to some existing FICs. In addition, the evolution can include modifications to some existing hook descriptions. Introducing new FICs requires generating test cases for them from scratch. Introducing new functionalities to some existing FICs requires either augmenting some existing test cases or creating test cases

from scratch. In the former case, an effective way is needed to identify the test cases to be augmented and then to augment them. Finally, modifying existing hook descriptions requires identifying the affected test cases and modifying them.

7.2.3. Using framework test cases

FICs can extend some framework extensible classes. In our research, we have looked at generating test cases for the FICs that extend framework classes using hook descriptions. When application developers add new methods and attributes to the FICs to implement the application requirements, new states and transitions can be added to the state-transition models of the FICs. Therefore, some reusable test cases have to be augmented or some new test cases have to be created from scratch to cover the new specifications.

The same scenario can be applied for the framework extensible classes at the framework development stage. FICs can introduce new methods and attributes to be used with the ones inherited from the extensible framework classes. Therefore, instead of generating the test cases for FICs from scratch, test cases generated for the framework extensible classes can be augmented to cover portion of the specifications of the FICs and new test cases can be added to cover the rest of the specifications. At the application development stage, the test cases can be used as illustrated in this research.

To apply the latter technique, it is required to introduce a technique to integrate the technique used in generating the framework test cases with the all paths-state technique used to solve the ignored specification problem. In our work, the augmentation of the test cases is delayed until the application development stage where all paths-state coverage is no longer needed because once the application is developed no specifications are ignored unless the application is modified.

7.2.4. Conducting more case studies

The experiments conducted in this research are valuable in showing the practicality of the introduced approaches and in studying the relationship between the framework type

and the reusability of the test cases. However, we believe that more experiments should be performed to confirm (or disprove) our results. In addition, more experiments can be performed to directly measure the saving in application testing time when using the reusable test cases provided with the framework.

7.2.5. Building reusable test cases at product line stages

In our research, framework application reusable test cases are generated at the framework development stage. When framework applications are developed as part of a product line and test cases are built to test the products, the test cases can be stored in a database each time an application is developed. Whenever another product is developed, the applicable test cases stored in the database are reused. In this case, the testing cost of the first product is high, but the testing cost is expected to decrease gradually as more products are developed.

The idea can be enhanced by combining it with the idea of generating test cases at the framework development stage introduced in this thesis. In this case, the reusable test cases are generated at the framework development stage as proposed in this thesis and stored in a reusable test case database. Whenever a product is developed, the new test cases are also stored in the reusable test case database, which can reduce the testing cost for the following product line applications. In this case, the testing cost of the first product is reduced because of the reusable test cases provided with the framework and the testing costs of the following products are reduced much more as the number of considered applications in the product line increases.

7.2.6. Extending the supporting tool

The prototype version of the $FIST_2$ tool generates test cases at the framework development stage using a state-transition table developed manually by the user by applying the synthesizing algorithms for the FIC models as explained in Section 3.2. The synthesizing algorithms are labor-intensive and, therefore, implementing them enhances the tool. The second possible extension of the tool is to support the generation of the test data and the code required to satisfy the predicates of the transitions. This generation is

performed manually in the prototype version of the tool and takes considerable time for complex state-based models. The third possible extension of the tool is to integrate it with the Hook Master tool. In this case, the $FIST_2$ tool gets updates on the FIC specifications introduced by the application developer directly from the Hook Master tool while the application is being developed. The $FIST_2$ tool can then update the state-transition table of the FIC automatically. This process is performed manually in the prototype version of the tool and took considerable effort and time in the conducted case studies. The fourth possible extension of the tool is to determine the required stubs by analyzing the hook descriptions and to generate the stubs. Finally, the $FIST_2$ tool can be integrated with other tools such as JTest [JTest], a supporting tool for method-based testing, and Test Mentor [Test Mentor] and Jverify [Jverify], tools for supporting integration-based testing, to form a Java testing environment in which FICs are tested at the method, class, and cluster levels. To use the JTest tool it is required to build a FIC mapping class that includes the declarations of all methods introduced by the hooks and instrument the class with the specifications of the FIC methods written in DbC. To use the Jverify or Test Mentor tool, it is required to write a piece of code to force the tool to use the class-based test cases generated using the $FIST_2$ tool instead of using the class-based test cases generated by Jverify or Test Mentor.

7.2.7. Generating and using reusable cluster-based test cases

The most important extension of the thesis is in the area of generating and using reusable cluster-based test cases. At the class level testing, stubs are required to isolate the class under test from other classes. From this point, the classes are added and tested gradually. There are several forms of interactions between classes including method invocations and global data sharing. Method invocations can be direct or indirect by invoking a method that directly or indirectly invokes a method of another object. These forms of interactions can exist in the methods defined in the hooks. These methods also can access global data accessed by methods of other classes defined in the hooks or in the framework. There are three areas related to the FICs for which cluster testing can be applied.

1. Integration between the FICs and the framework classes

Hooks can define methods that invoke methods of the framework classes directly (e.g., the invocation statement in Line 13 of the code listed in Figure 7.1) or indirectly (e.g., invoking *Amethod* declared in Line 8 of the code listed in Figure 7.1 indirectly by the invocation statement in Line 13). The same situation applies for the methods of the framework classes, where these methods can invoke the methods defined in the hooks (e.g., the invocation of *Amethod* declared in Line 25 of the code listed in Figure 7.1 when the statement in Line 21 is executed). Moreover, the hooks can define methods that share the same global data with the methods of the framework classes (e.g., *globalVar* shared between the methods declared in Lines 3 and 12 of the code listed in Figure 7.1). If any of these forms of integration exist, cluster testing is required to test the interactions between the FICs and the framework classes.

```
1   public AframeworkClass {      // a framework class
2       public static int globalVar;
3       public AframeworkClass() {
4           Amethod();
5           globalVar = 1;
6           …
7       }
8       public void Amethod() { … }
9       …
10  }
11  public FICA {          //FIC
12      public FICA() {
13          AframeworkClass var = new AframeworkClass();
14          var.globalVar = globalVar;
15          …
16      }
17       …
18  }
19  public FICB extends AframeworClass {     //FIC
20      public FICB() {
21          super();
22          FICA var = new FICA();
23           …
24      }
25      public void Amethod() { … }
26       …
27  }
28  public Aclass {          //Application class which is not FIC
29      public Aclass {
30          FICA var = new FICA();
31           …
32      }
33       …
43  }
```

Figure 7.1: Java code example

2. Integration among the FICs

Hooks can define methods that invoke methods of other FICs directly (e.g., the invocation statement in Line 22 of the code listed in Figure 7.1) or indirectly. Moreover, hooks can define methods that share the same global data with the methods of other FICs. If any of these forms of integration exist, cluster testing is required to test the interactions among the FICs.

3. Integration between the FICs and other application classes

Application developers can add methods to the FICs and they also can add new classes to the framework applications. The code of the new classes can include invocations for methods of the FICs (e.g., the invocation statement in Line 30 of the code listed in Figure 7.1) or share global data accessed by the FICs. Moreover, the new methods added to the FICs at the application development stage can invoke the methods of the added classes or access global data accessed by the added classes. If any of these forms of integration exist, cluster testing is required to test the interactions between the FICs and the added classes.

- Discussion

The knowledge about the integrations among the FICs and between the FICs and the framework classes is defined in the hooks. Therefore, reusable cluster-based test cases that test the integration among the FICs and between the FICs and the framework classes can be built at the framework development stage and used at the cluster testing step of the applications developed using the framework. However, the knowledge about the integration between the FICs and the other application classes does not exist at the framework development stage because the other application classes are not known at the framework development stage. Therefore, no reusable cluster-based test cases can be built at the framework development stage to test the integration between the FICs and the other application classes.

There are several problems to be solved to build and use the reusable cluster-based test cases that test the integration among the FICs and between the FICs and the framework classes. The first problem is determining the suitable testing models that can be used to build the test cases. Typically, researchers use the class, sequence, and collaboration diagrams to generate cluster-based test cases (e.g., [Binder 99, Badri+ 02, Wu+ 03]). The second problem is showing how to use the testing models in building the reusable cluster-based test cases that have high coverage for the cluster relationships implemented in the framework applications. The problem can be solved by merging an existing cluster-based testing technique with the all paths-state coverage idea. The third problem requires studying the affect of the modifications (e.g., ignoring or adding specifications) performed at the application development stage on the reusable cluster-based test cases.

References

[Al Dallal+ 97] J. Al Dallal and K. Saleh, Synthesis of distributed and concurrent protocol systems, *Proc. Of the 15th Intern. Conf. On Information Super Highway Trends and Impact, Saudi Arabia*, November 1997, pp.665-678.

[Al Dallal+ 02] J. Al Dallal and P. Sorenson, System testing for object-oriented frameworks using hook technology, *Proc. of the 17th IEEE International Conference on Automated Software Applications (ASE'02)*, Edinburgh, UK, September 2002, pp. 231-236.

[Antoniol+ 02] G. Antoniol, L. Briand, M. Penta, and Y. Labiche, A case Study Using the Round-Trip Strategy for State-based Class Testing, *Carlton University TR SCE-01-08*, revised Jan. 2002.

[Abdurazik+ 00a] A. Abdurazik, P. Ammann, W. Ding, and J. Offutt, Evaluation of three specification-based testing criteria, *Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '00)*, Tokyo, Japan, September 2000, pp 179-187.

[Abdurazik+ 00b] A. Abdurazik and J. Offutt, Using UML collaboration diagrams for static checking and test generation, *Proc. Of The Third International Conference on the Unified Modeling Language (UML '00), York, UK*, Oct. 2000, pp. 383-395.

[Badri+ 02] L. Badri and M. Badri, Test sequences generation from UML collaboration diagrams: towards a formal approach, *IASTED International Conference on Software Engineering and Applications (SEA 2002)*, Cambridge, USA, November, 2002, pp. 477-483.

[Ball+ 00] T. Ball, D. Hoffman, F. Ruskey, R. Webber, and L. White, State generation and automated class testing, *Software Testing, Verification and Reliability*, (10) 2000, pp. 149-170.

[Baudry+ 01] B. Baudry, Y. LeTraon, and J.-M. Jézéquel, Robustness and diagnosability of OO-systems designed by contracts, *Proceedings of Metrics'01*, London, UK, April 2001, pp. 272-283.

[Beck+ 94] K. Beck and R, Johnson. Patterns generate architectures, *Proc. of ECOOP 94*, 1994, pp. 139-149.

[Beizer 90] B. Beizer. *Software testing techniques*, 2nd ed. New York: International Thompson Computer Press, 1990.

[Beizer 95] B. Beizer. *Black-Box Testing*, John Wiley, 1995.

[Beugnard+ 99] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, Making components contract aware, *IEEE Computer*, 13(7), July 1999, pp. 38-45.

[Beydeda] S. Beydeda, V. Gruhn, and M. Stachorski, A graphical class representation for integrated black- and white-box testing, *IEEE International Conference on Software Maintenance (ICSM'01)*, Florence, Italy, November, 2001, pp. 706-715.

[Binder 96a] R. Binder. Testing object-oriented software: A survey, *Software Testing, Verification and Reliability*, 6(3/4), 1996, 125-252.

[Binder 96b] R. Binder. Testing for reuse: libraries and frameworks, *Object Magazine*, August 1996, 77-80.

[Binder 99] R. Binder. *Testing object-oriented systems*, Addison Wesley, 1999.

[Bogdanov+ 01] K. Bogdanov and M. Holcombe, Statechart testing method for aircraft control systems, *Software Testing, Verification and Reliability*, Vol. 11(1), 2001, pp. 39-54.

[Boujarwah+ 00] A. Boujarwah, K.Saleh, and J. Al-Dallal. Dynamic Data Flow Analysis for Java Programs, *Journal of Information and Software Technology,* Vol. 42, No. 11, August 2000, pp. 765-775.

[Boyapali+ 02] C. Boyapati, S. Khurshid, and D. Marinov, Korat: Automated Testing Based on Java Predicates, *International Symposium on Software Testing and Analysis ISSTA*, Rome, Italy, July 2002, pp. 123-133.

[Briand+ 02a] L. Briand and Y. Labiche, A UML-based approach to system testing, *Carlton University TR SCE-01-01*, revised February 2002.

[Briand+ 02b] L. Briand, Y. Labiche, and H. Sun, Investigating the use of analysis contracts to support fault isolation in object-oriented code, *International Symposium on Software Testing and Analysis ISSTA*, Rome, Italy, July 2002, pp. 70-80.

[Calliss+ 88] F. W. Calliss and B.J. Cornelius. Dynamic data flow analysis of C programs, *Proceedings of the 21st annual Hawaii international conference*, Vol. 2, 1988, pp. 518-523.

[Campbell+ 92] R. H. Campbell and N. Islam. A Technique for documenting the framework of an object-oriented system, *Proceeding of the 2nd International Workshop on the Object-Oriented in Operating Systems, Paris, France*, 1992,

[Chan+ 87] F.T. Chan and T.Y. Chen. AIDA-a dynamic data flow anomaly detection system for Pascal programs, *Software-Practice and Experience*, Vol. 17, 1987, pp. 227-239.

[Chen+ 87] T. Y. Chen, H. Kao, M.S. Luk, and W.C. Ying. COD-a dynamic data flow analysis system for Cobol, *Information and Management*, Vol. 12, Feb 1987, pp. 65-72.

[Chen+ 95] T. Y. Chen and C.K. Low. Dynamic data flow analysis for C++, *Proceeding of 1995 Asia Pacific Software Engineering Conference*, 1995, pp. 22-28.

[Chen+ 98] H. Chen, T. Tse, F. Chan, and T. Chen, In black and white: an integrated approach to class-level testing of object-oriented programs, *ACM Transactions on Software Engineering and Methodology*, Vol. 7, No. 3, July 1998, pp. 250-295.

[Chen+ 01] H. Chen, T. Tse, and T. Chen, TACCLE: a methodology for object-oriented software Testing At the Class and Cluster Levels, ACM Transactions on Software Engineering and Methodology, Vol.10, No.1, Jan. 2001, pp.56-109

[Cheon+ 02] Y. Cheon and G. Leavens, A simple and practical approach to unit testing: the JML and JUnit way, *Proc. of the 16th European Conference on Object-Oriented Programming (ECOOP2002)*, June 2002, pp. 231-254.

[Chow 78] T. Chow, Testing software design modeled by finite state machines, *IEEE Transactions on Software Engineering*, Vol. EE-4(3), 1978, pp. 178-187.

[Codenie+ 97] W. Codenie, K. De Hondt, P. Steyaert, and A. Vercammen. From custom applications to domain-specific frameworks. *Communications of the ACM*, Vol. 40(10), October 1997. pp. 71-77.

[CSF] CSF: Client-Server Framework, http://www.cs.ualberta.ca/~garry/framework, December 2001.

[Daley+ 02] N. Daley, D. Hoffman, and P. Strooper, A framework for table driven testing of Java classes, *Software-Practice and Experience*, 32, 2002, pp. 465-493.

[Doong+ 94] R. Doong and P. Frankl, The ASTOOT approach to testing object-oriented programs, *ACM Transactions on Software Engineering and Methodology*, Vol. 3, No. 2, April 1994, pp. 101-130.

[Fayad+ 97] E. M. Fayad and D. C. Schmidt. Object-oriented application frameworks, *Communications of the ACM*, October 1997, Vol. 40, No. 10, pp. 32-38.

[Fenkam+ 02] P. Fenkam, H. Gall and M. Jazayeri, Constructing corba-supported oracles for testing: a case study, *Proc. of the 17th IEEE International Conference on Automated Software Applications (ASE'02)*, Edinburgh, UK, September 2002, pp. 129-138.

[Froehlich 02] G. Froehlich, Hooks: an aid to the reuse of object-oriented frameworks, Ph.D. Thesis, University of Alberta, Department of Computing Science, 2002.

[Froehlich+ 97] G. Froehlich, H.J. Hoover, L. Liu, and P.G. Sorenson. Hooking into Object-Oriented Application Frameworks, *Proc. 19th Int'l Conf. on Software Engineering*, Boston, May 1997, pp. 491-501.

[Froehlich+ 98] G. Froehlich, H.J. Hoover, L. Liu, and P.G. Sorenson, Using Object-Oriented Frameworks, *CRC Handbook of Object Technology, CRC Press*, 1998, pp. 26-1 - 26-22.

[Gamma+ 95] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, Reading, MA, 1995.

[Gangopadhyay+ 95] D. Gangopadhyay and S. Nitra. Understanding frameworks by exploration of exemplars, *Proceedings of 7th International Workshop on Computer Aided Software Engineering (CASE-95), Toronto, Canada*, 1995, pp. 90-99.

[Harrold+ 92] M. Harrold, J. McGregor, and K. Fitzpatrick, Incremental testing of object-oriented class structures, *Proc of the 14th international conference on Software Engineering*, 1992, pp. 68-80.

[Harrold+ 01] M. Harrold, J. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. Spoon, and A. Gujarathi, Regression test selection for Java software, *Proc. of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2001)*, Tampa, Florida, USA, October 2001, pp. 312-326.

[Hoffman+ 94] D. Hoffman and P. Strooper, Graph-based class testing, *The Australian Computer Journal*, Vol. 26, No. 4, Nov. 1994, pp. 158-163.

[Hoffman+ 97] D. Hoffman and P. Strooper, Classbench: a framework for automated class testing, *Software-Practice and Experience*, 27(5), 1997, pp. 573-597.

[Hong+ 00] H. Hong, Y. Kim, S. Cha, D. Bae, H. Ural, A test sequence selection method for statecharts, *Software Testing, Verification and Reliability*, 10/4, 2000, pp. 203-227.

[Hornstein+ 02] J. Hörnstein, H. Edler, Test reuse in CBSE using built-in tests, *Component-based Software Engineering Workshop: Composing Systems From Components, (ECBS 2002)*, Lund, Sweden. April 2002.

[Hsia+ 97] P. Hsia, X. Li, D. Kung, C. Hsu, L..Li, Y.Toyoshima, and C.Chen, A technique for the selective revalidation of OO software, *Journal of Software Maintenance*, Vol. 9, 1997, pp.217-233.

[iContract] iContract: the Java Design-by-Contract tool, http://www.reliable-systems.com/tools/ iContract/iContract.htm, September, 2002.

[IEEE 829] *ANSI/IEEE standard 610.12-1990: glossary of software engineering terminology*. New York: The Institute of Electrical and Electronic Engineers, 1987.

[Jcontract] Jcontract, http://www.parasoft.com/jsp/products/home.jsp?product= Jcontract, ParaSoft Corporation, April 2002.

[Johnson 92] R. Johnson. Documenting frameworks using patterns, *Proceedings of OOPSLA'92, Vancouver, Canada*, 1992, pp. 63-76.

[Johnson+ 88] R. Johnson and B. Foote. Designing reusable classes, Journal of Object-Oriented Programming, Vol. 2(1), 1988, pp.22-35.

[JTest] Jtest, http://www.parasoft.com/jsp/products/home.jsp?product=Jtest, ParaSoft Corporation, September 2002.

[Junit] Junit, http://junit.sourceforge.net, December 2001.

[Jverify] JVerify, http://software.qip.us/jverify.htm, January, 2003

[Kim+ 99] Y. Kim, H. Hong, D. Bae, and S. Cha, Test cases generation from UML state diagrams, *IEE Proc.-Software*, Vol. 146, No. 4, 1999, pp. 187-192.

[Krasner+ 88] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80, *Journal of Object-Oriented Programming*, 1(3), August-September 1988, 26-49.

[Kung+ 94a] Kung, D., J.Gao, P.Hsia, Y.Toyoshima, and C.Chen, Firewall regression testing and software maintenance of object-oriented systems, Journal of Object-Oriented programming, 1994.

[Kung+ 94b] Kung, D., J.Gao, P.Hsia, F.Wen, Y.Toyoshima, and C.Chen, Change impact identification in object oriented software maintenance, *Proc. IEEE International Conference on Software Maintenance*, 1994, pp. 202 – 211.

[Kung+ 96] Kung, D., J.Gao, P.Hsia, F.Wen, Y.Toyoshima, and C.Chen, On regression testing of object-oriented programs, The Journal of Systems and Software, 32(1), January 1996, pp. 21-40.

[Lajoie+ 94] R. Lajoie and R. K. Keller. Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert, *Proceedings of the 62nd Congress of the Association Canadienne Francaise pour l'Avancement des Science, Montreal, Canada,* 1994.

[Leavens+ 99] G. Leavens, A. Baker, and C. Ruby, JML: a notation for detailed design. *In H. Kilov, B. Rupe, and I. Simmonds, editors, behavioral specifications of Businesses and Systems, chapter 12, Kluwer*, 1999, pp. 175-188.

[Leavens+ 01] G. Leavens, A. Baker, and C. Ruby, Preliminary design of JML: a behavioral interface specification language for Java, *TR 98-06p, Iowa State University, Department of Computer Science*, August 2001.

[LOCC] LOCC (software information), http://csdl.ics.hawaii.edu/Tools/LOCC/LOCC.html, October 2002.

[McDonald+ 96] J. McDonald and P. Strooper, Testing inheritance hierarchies in the ClassBench framework, *Proc. of the Technology of Object-Oriented Languages and Systems (TOOLS USA '96)*, August 1996.

[McDonald+ 97] S. McDonald, J. Schaeffer, and D. Szafron. Pattern-based object-oriented parallel programming. *Proceedings of the First International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97)*, Vol. 1343 of Lecture Notes in Computer Science, 1997, pp. 167-274.

[McGregor 00] J. McGregor, Building reusable test assets for a product line tutorial, *First Software Product Line Conference*, Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 2000.

[McGregor 01] J. McGregor, Testing a software product line, *Technical Report CMU/SEI-2001-TR-022*, Software Engineering Institute, Pittsburgh, PA, December 2001.

[Meyer 92] B. Meyer, Design by contracts, *IEEE Computer*, 1992, Vol. 25(10), 40-52.

[Myers 79] Myers. *The Art of Software Testing*, John Wiley, 1979.

[Murray+ 97] L. Murray, D. Carrington, I. MacColl, and P. Strooper, Extending test templates with inheritance, *Proc. of the Australian Software Engineering Conference (ASWEC '97)*, September 1997, pp. 80-87.

[NewCAO Model] http://www.cs.ualberta.ca/~jehad/CAO_model.xls

[Offutt+ 97] J. Offutt and J. Pan, Automatically detecting equivalent mutants and infeasible paths, *The Journal Of Software Testing, Verification, and Reliability*, 7(3), September 1997, pp 165-192.

[Offut+ 99] J. Offut and A. Abdurazik, Generating tests from UML specifications, *Second International Conference on the Unified Modeling Language (UML99)*, Fort Collins, CO, October 1999, 416-429.

[Osterweil+ 78] L. J. Osterweil and L. D. Fosdick. DAVE-a validation error detection and documentation system for FORTRAN programs, *Computer*, Vol. 11, 1978, pp. 25-32.

[Price+ 85] D. A. Price. Program instrumentation for the detection of software anomalies, *M. Sc. Thesis, Department of computer science, University of Melbourne*, 1985.

[Roper 94] M. Roper. *Software Testing*, McGraw-Hill, 1994.

[Rothermel+ 94] G. Rothermel and M. Harrold, Selecting regression tests for object-oriented software, *Proc. IEEE International Conference on Software Maintenance*, 1994, pp. 14-25.

[Rothermel+ 00] G. Rothermel, M. Harrold, and J. Dedhia, Regression test selection for C++ softawrem Journal of software testing, Verification, and Reliability, 10(6), January 2000, pp. 77-109.

[Saleh+ 01] K. Saleh, A. Boujarwah and J. Al-Dallal, "Anomaly detection in concurrent Java programs using dynamic data flow analysis", *Journal of Information and Software Technology*, Dec. 2001, Vol 43, No. 15, pp. 973-981.

[SalesPoint] The SalesPoint Framework V2.0 Homepage, http://ist.unibw-muenchen.de /Lectures/SalesPoint/, November 2002.

[SalesPoint applications] SalesPoint applications, http://ist.unibw-muenchen.de/Lectures/ HT2001/Praktikum/Applications.htm, November 2002.

[Sparks+ 96] S. Sparks, K. Benner, and C. Faris. Managing object-oriented frameworks reuse. IEEE Computer, Vol 29(9), September 1996, pp. 52-61.

[Swing] Swing, Java 1.3.1, http://java.sun.com/j2se/1.3/, October 2002.

[Test Mentor] Test Mentor - Java Edition by SilverMark, http://www.componentsource. com/ProductCatalog/TestMentorJavaEdition.htm, January, 2003.

[Tsai+ 99] W. Tsai, Y. Tu, W. Shao, and E. Ebner. Testing extensible design patterns in object-oriented frameworks through scenario templates, *23rd Annual International Computer Software and Applications Conference, Phoenix, Arizona*, October, 1999, pp. 166-171.

[WaveFront Documentation] WaveFront Documentation, http://www.cs.ualberta.ca/ ~janvik/thesis/Document.html, December 2001.

[Wang+ 00] Y. Wang, D. Patel, G. King, I. Court, G. Staples, M. Ross, and M. Fayad, On built-in test reuse in object-oriented framework design, *ACM Computing Surveys (CSUR)*, Vol. 32(1es), March 2000, pp. 7-12.

[White+ 97] L. White and K. Abdullah, A firewall approach for regression testing of object-oriented software, Proc. of the 10th Annual Software Quality Week, May 1997.

[Wilkin+ 02] S. Wilkin and D. Hoffman. JUnit extensions for documentation and inheritance, *Proc. of the 2002 Pacific Northwest Software Quality Conference*, Protland, USA, October 2002, pp. 71-84.

[Wu+ 03] Ye Wu, Mei-Hwa Chen and Jeff Offutt, UML-based integration testing for component-based software, *The 2nd International Conference on COTS-Based Software Systems (ICCBSS '03)*, Ottawa, Canada, February 2003, 251-260.

Appendix A Example for constructing the all paths-state tree for a model free of guaranteed transitions

Appendix A illustrates the steps of the procedure given in Figure 3.8 to construct the all paths-state tree for the STD of the *NewAccount* FIC shown in Figure 3.4. First the root node represents the Alpha state of the STD. In the first iteration of the repeat loop of the procedure, an edge and a node are drawn to represent the outgoing transition from the Alpha state and the destination state of the transition as shown in Figure A.1. Each time an edge is drawn, the label associated with the transition is represented on the edge.

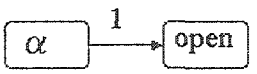

Figure A.1: The results of applying the first iteration of the repeat loop of the procedure given in Figure 3.8 for the STD shown in Figure 3.4

In the second iteration of the repeat loop, the outgoing transitions from the Open state are represented in the tree by edges as shown in Figure A.2. In the figure, bolded nodes represent terminal nodes (i.e., no more edges are drawn from these nodes). Figure A.2 shows four terminal nodes. These nodes are marked terminal because they are either encountered on the tree paths that contain them or they represent the omega state. Figures A.3 and A.4 show the results of applying the third and fourth iteration of the loop, respectively. All the leaf nodes in Figure A.4 are marked terminal, which indicates that the construction process of the tree is completed.

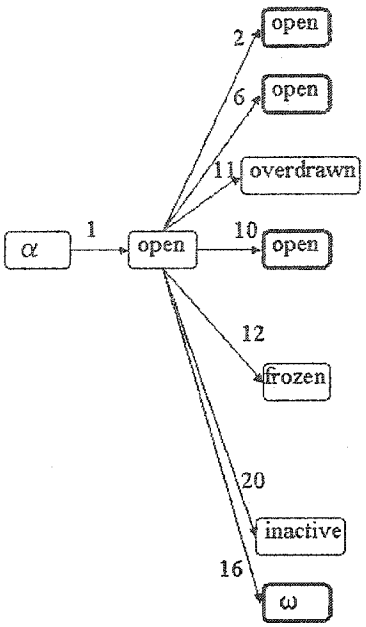

Figure A.2: The results of applying the second iteration of the repeat loop of the procedure given in Figure 3.8 for the STD shown in Figure 3.4

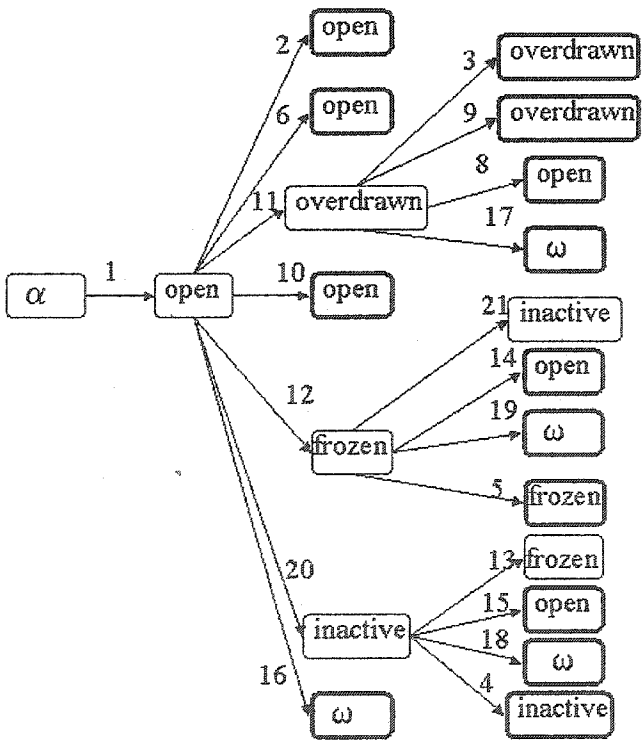

Figure A.3: The results of applying the third iteration of the repeat loop of the procedure given in Figure 3.8 for the STD shown in Figure 3.4

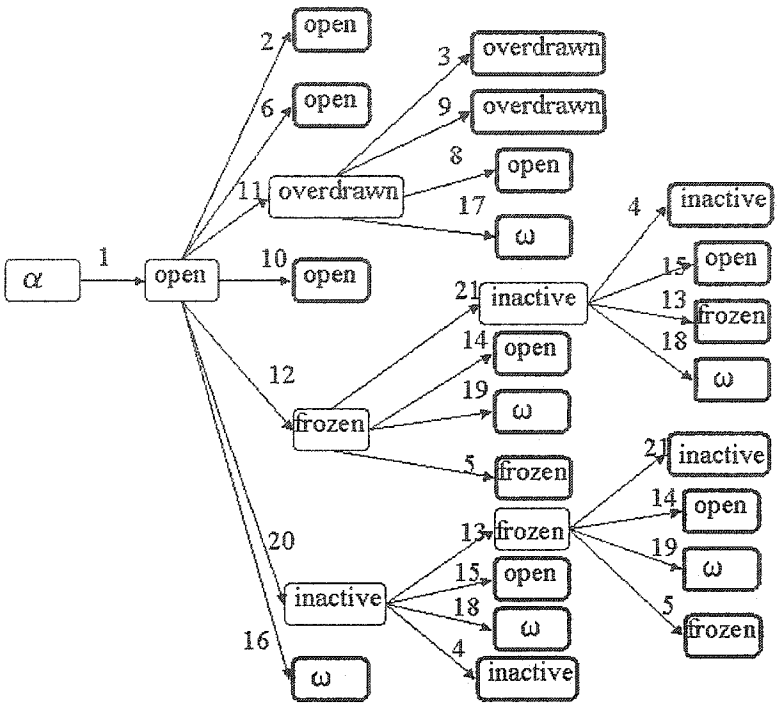

Figure A.4: The results of applying the fourth iteration of the repeat loop of the procedure given in Figure 3.8 for the STD shown in Figure 3.4

Appendix B Example for constructing the all paths-state tree for a model that has guaranteed transitions

Appendix B illustrates the steps of the procedure given in Figure 3.12 to construct the all paths-state tree for the STD of the *NewAccount* FIC shown in Figure 3.4 assuming that the transitions necessary to implement the open and inactive states (i.e., the transitions labeled by 1, 2, 4, 6, 10, 15, 16, 18, and 20) of Figure 3.4 are marked as guaranteed. First, the root node represents the Alpha state of the STD. In the first iteration of the repeat loop of the procedure, an edge and a node are drawn to represent the outgoing transition from the Alpha state and the transition destination state as shown in Figure B.1. The two nodes drawn in the first iteration are bolded. A bolded node and a bolded edge represent a guaranteed node and a guaranteed edge, respectively. The *alpha* node is always marked guaranteed. The outgoing edge from the *alpha* node is bolded because it represents a guaranteed transition. Finally, the *open* node is marked guaranteed because it is a destination node of a guaranteed edge initiated from a guaranteed node.

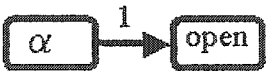

Figure B.1: The results of applying the first iteration of the repeat loop of the procedure given in Figure 3.12 for the STD shown in Figure 3.4

In the second iteration of the repeat loop, the outgoing transitions from the Open state are represented in the tree by edges as shown in Figure B.2. In the tree drawn so far in Figure A.2, nodes reached by the edges labeled by 2, 6, 10, and 16 are marked terminal (i.e., no more edges are drawn from them) because they are either previously encountered on the tree paths that contain them or they represent the omega state. In addition, nodes reached by the edges labeled by 2, 6, 10, 20 and 16 are marked guaranteed because they are destination nodes of guaranteed edges initiated from a guaranteed node.

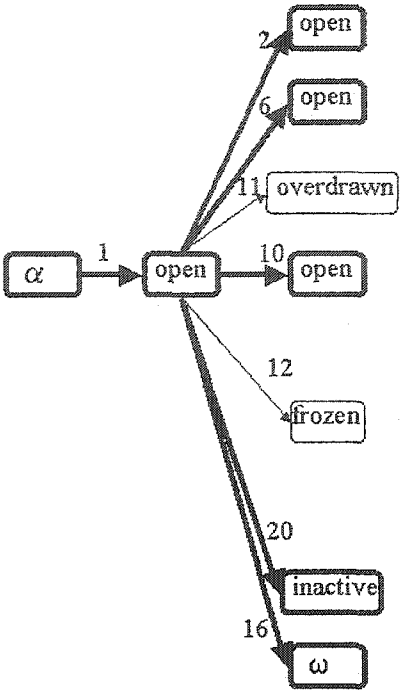

Figure B.2: The results of applying the second iteration of the repeat loop of the procedure given in Figure 3.12 for the STD shown in Figure 3.4

The tree shown in Figure B.2 contains one node marked as non-terminal guaranteed, which is the node that represents the Inactive state. Therefore, in the third iteration of the repeat loop, edges that represent the outgoing transitions from Inactive state and the nodes that represent the states reached from the Inactive state are drawn as shown in Figure B.3. Three of the drawn nodes in the third iteration (i.e., open, omega, and inactive) are marked terminal because they are either previously encountered on the tree paths that contain them or they represent the omega state. In addition, these nodes are marked guaranteed because they are destination nodes of guaranteed edges initiated from a guaranteed node. The fourth node (i.e., frozen) is marked non-terminal and not-guaranteed.

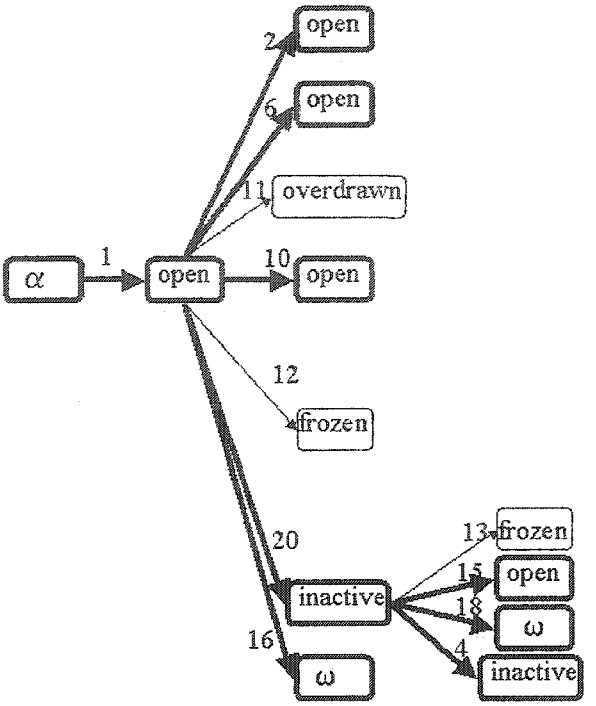

Figure B.3: The results of applying the third iteration of the repeat loop of the procedure given in Figure 3.12 for the STD shown in Figure 3.4

All the non-terminal leaf nodes drawn so far in Figure B.3 are marked not-guaranteed. Therefore, in the fourth iteration we can pick any of the states that represent each of them. In the example, we picked the node that represents the overdrawn state. In the fourth iteration, edges that represent the outgoing transitions from the overdrawn state and the nodes that represent the states reached from the overdrawn state are drawn as shown in Figure B.4. All the drawn nodes are marked terminal because they are either previously encountered on the tree paths that contain them or they represent the omega state. In addition, these nodes are marked not-guaranteed because they are reached by not-guaranteed edges.

All the non-terminal leaf nodes drawn so far in Figure B.4 are marked not-guaranteed. Therefore, in the fifth iteration, we can pick any of the states that represent each of them. In the example, we picked the node that represents the frozen state reached by the edge labeled as 12. Edges that represent the outgoing transitions from the frozen state and the nodes that represent the states reached from the frozen state are drawn as shown in Figure

B.5. Three of the drawn nodes (i.e., open, omega, and frozen) are marked terminal because they are either previously encountered on the tree paths that contain them or they represent the omega state. The fourth drawn node (i.e., inactive) is marked terminal because it represents a state represented in the tree by a guaranteed node. The four nodes are marked not-guaranteed because they are reached by not-guaranteed edges.

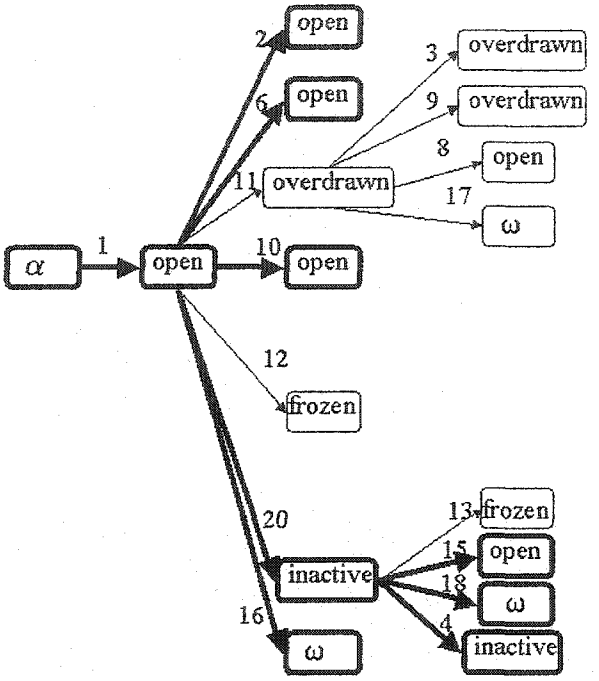

Figure B.4: The results of applying the fourth iteration of the repeat loop of the procedure given in Figure 3.12 for the STD shown in Figure 3.4

Only one leaf node in the tree (i.e., frozen) shown in Figure B.5 is marked non-terminal. Edges that represent the outgoing transitions from the frozen state and the nodes that represent the states reached from the frozen state are drawn in the sixth iteration as shown in Figure B.6. All the drawn nodes are marked terminal because they are either previously encountered on the tree paths that contain them or they represent the omega state. In addition, these nodes are marked not-guaranteed because they are reached by not-guaranteed edges. All the leaf nodes in Figure B.6 are marked terminal, which indicates that the construction process of the tree is completed.

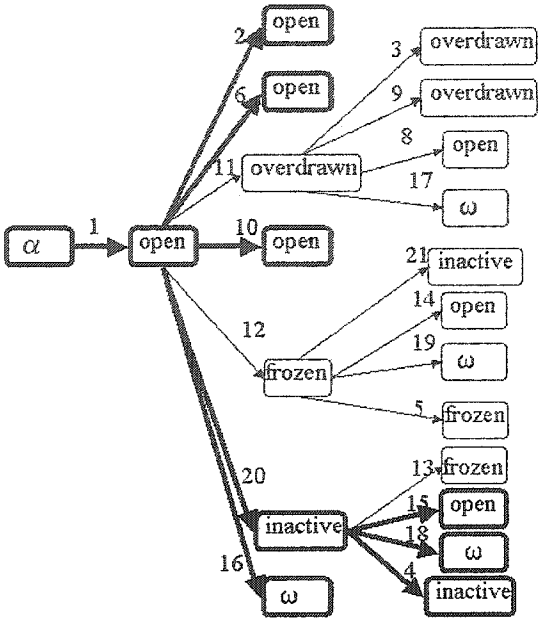

Figure B.5: The results of applying the fifth iteration of the repeat loop of the procedure given in Figure 3.12 for the STD shown in Figure 3.4

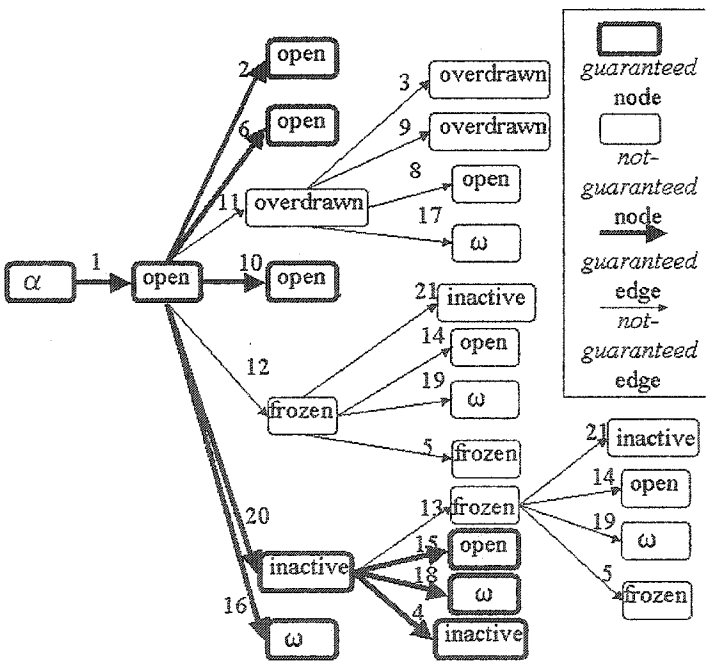

Figure B.6: The results of applying the sixth iteration of the repeat loop of the procedure given in Figure 3.12 for the STD shown in Figure 3.4

Appendix C The Syntax of the Testable State-transition Model Description

The $FIST_2$ tool uses the following syntax for the Testable State-transition Model Description (TSTMD) language to maintain the testable state-transition model description in a text file. An example for a use of the language is shown in Appendix D.

```
<Model_desc>:: <State_desc>[, .., <State_desc>]
               <State_trans>[, ..,<State_trans>]
<State_desc>:: <state_name>{
               <state_invariant>
               <app_specific>
              }
<state_name>::= <string>
<state_invariant>::= var_invariant = <string>
<app_specific>::= app_specific= <boolean>
<State_trans>::= <source_state> => <destination_state>{
                  <event>
                  [, ..,<parameter>]
                  [, ..,<predicate>]
                  [, ..,<action>]
                  <guaranteed>
                  <app_specific>
                  [<test_cases_covered>]
                  }
<source_state>::= <state_name>
<destination_state>::= <state_name>
<event>::= event = <string>
<parameter>::= parameter = <param_name>:<param_type>
<param_name>::= <string>
<param_type>::= <string>
<predicate>::= condition = <var><oper><var>;[(script)]
<var>::= <string>
<oper>::= > | < | = | <= | >=
<script>::= <string>
```

```
<action>::= action = <action_type>:<value>;[<script>]
<action_type>::= return | exception | message | others
<value> :: = <string>
<guaranteed> :: = guaranteed = <boolean>
<boolean> :: = true | false
<test_cases_covered> ::= [(, .., <test_case_id>)]
<test_case_id> ::= <string>
```

Appendix D MyAccount FIC Example

Appendix D shows the application of the $FIST_2$ tool for testing the *MyAccount* FIC class as follows.

D.1. Framework Development Stage

D.1.1. Tool Inputs

The following hook descriptions define the *NewAccount* FIC.

Name: Initialize Account
Requirement: Initialize an account (i.e., set the currency and bank branches).
...
Preconditions: amount>=0;
Changes:
NewAccount.NewAccount(int amount) extends Account.Account(int amount);
...
Postconditions:
1. Operation NewAccount. NewAccount (int);
2. NewAccount.balance()>=0;
3. ! NewAccount.isFrozen();
4. NewAccount. getCurrentDate()-NewAccount.getLastActivityDate() < NewAccount.MaxPeriod

...

Name: Get Account Balance
Requirement: inquire about the balance
...
Preconditions:
1. NewAccount subclass of Account;
Changes:
NewAccount.balance() reads supper.balance();
...
Postconditions:
1. Operation NewAccount. balance();

...

Name: Deposit Money
Requirement: deposit money in an account.
...
Preconditions:
1. NewAccount subclass of Account
2. ! NewAccount.isFrozen();

3. NewAccount. getCurrentDate()-NewAccount.getLastActivityDate() < NewAccount.MaxPeriod

Changes:

NewAccount.deposit(int amount) extends Account.deposit(int amount);

...

Postconditions:

1. Operation NewAccount. deposit (int);
2. NewAccount.balance()=amount+ NewAccount.balance();
3. ! NewAccount.isFrozen();
4. NewAccount. getCurrentDate()-NewAccount.getLastActivityDate() < NewAccount.MaxPeriod

...

Name: Withdraw Money

Requirement: withdraw money from an account.

...

Preconditions:

1. NewAccount subclass of Account
2. NewAccount.balance()>=0
3. ! NewAccount.isFrozen();
4. NewAccount. getCurrentDate()-NewAccount.getLastActivityDate() < NewAccount.MaxPeriod

Changes:

NewAccount.withdraw(int amount) extends Account.withdraw(int amount);

...

Postconditions:

1. Operation NewAccount. withdraw (int);
2. NewAccount.balance()=NewAccount.balance()-amount;
3. ! NewAccount.isFrozen();
4. NewAccount. getCurrentDate()-NewAccount.getLastActivityDate() < NewAccount.MaxPeriod

...

Name: Freeze Account

Requirement: Freeze an account.

...

Preconditions:

1. NewAccount subclass of Account
2. NewAccount.balance()>=0
3. ! NewAccount.isFrozen();

Changes:

New Operation NewAccount.freeze();

...

Postconditions:

1. Operation NewAccount. freeze ();
2. NewAccount.balance()>=0;
3. NewAccount.isFrozen();

...

Name: Unfreeze Account
Requirement: Unfreeze an account.
...
Preconditions:
1. NewAccount subclass of Account
2. NewAccount.balance()>=0
3. NewAccount.isFrozen();

Changes:
New Operation NewAccount.unfreeze();
...

Postconditions:
1. Operation NewAccount. unfreeze ();
2. NewAccount.balance()>=0;
3. ! NewAccount.isFrozen();
4. NewAccount. getCurrentDate()-NewAccount.getLastActivityDate() < NewAccount.MaxPeriod

...

Name: Activate Account
Requirement: Activate an account.
...
Preconditions:
1. NewAccount subclass of Account
2. NewAccount.balance()>=0
3. ! NewAccount.isFrozen();
4. NewAccount. getCurrentDate()-NewAccount.getLastActivityDate() >= NewAccount.MaxPeriod

Changes:
New Operation NewAccount.activate();
...

Postconditions:
1. Operation NewAccount. activate ();
2. NewAccount.balance()>=0;
3. ! NewAccount.isFrozen();
4. NewAccount. getCurrentDate()-NewAccount.getLastActivityDate() < NewAccount.MaxPeriod

...

In the current prototype version of the $FIST_2$ tool, we are not yet able to use the hook descriptions to synthesize automatically the FIC model. Instead the user has to create the FIC model manually as illustrated in Section 3.2 and interacts with the tool to fill a table that describes the model. Figure D.1 shows the filled table for the *NewAccount* FIC.

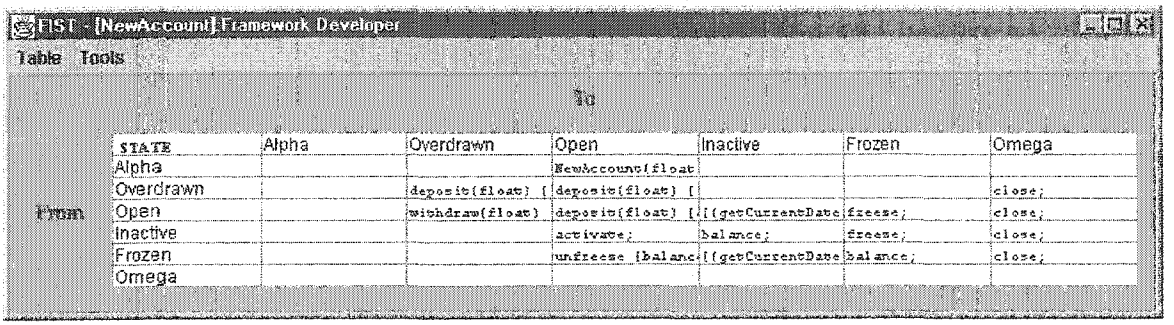

Figure D.1: The tabular form of the *NewAccount* FIC

The $FIST_2$ tool translates the tabular form of the model to the following model description form using the TSTMD language. The Java implementations of the predicates are associated with the transitions.

```
"Alpha"{
        var_invariant = ""
        app_specific = false
        }

"Overdrawn"{
        var_invariant = "(o.balance()<0) && ((o.getCurrentDate()-
         o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())"
        app_specific = false
        }

"Open"{
        var_invariant = "(o.balance()>=0) && ((o.getCurrentDate()-
         o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())"
        app_specific = false
        }

"Inactive"{
        var_invariant = "(o.balance()>=0) && ((o.getCurrentDate()-
         o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())"
        app_specific = false
        }

"Frozen"{
        var_invariant = "(o.balance()>=0) && (o.isFrozen())"
        app_specific = false
        }

"Omega"{
        var_invariant = "((o.balance()<0.01)&&(o.balance()>-0.01))"
        app_specific = false
        }

"Alpha"=>"Open"{
            event = "NewAccount"
```

```
            parameter = "amount":"float"
            condition = "amount" >= "0";[("float amount=1;")]
            guaranteed = true
            app_specific = false
            }
"Overdrawn"=>"Overdrawn"{
            event = "deposit"
            parameter = "amount":"float"
            condition = "(balance + amount)" < "0";[("amount=
                        o.balance();")]
            guaranteed = true
            app_specific = false
            }
"Overdrawn"=>"Overdrawn"{
            event = "balance"
            guaranteed = true
            app_specific = false
            }
"Overdrawn"=>"Open"{
            event = "deposit"
            parameter = "amount":"float"
            condition = "(balance + amount)" >= "0";[("amount=1-
                        o.balance();")]
            guaranteed = true
            app_specific = false
            }
"Overdrawn"=>"Omega"{
            event = "close"
            guaranteed = true
            app_specific = false
            }
"Open"=>"Overdrawn"{
            event = "withdraw"
            parameter = "amount":"float"
            condition = "(balance - amount)" < "0";[("amount=1+
                        o.balance();")]
            guaranteed = true
            app_specific = false
            }
"Open"=>"Open"{
            event = "deposit"
            parameter = "amount":"float"
            condition = "amount" = "1.0";[("amount=1;")]
            action = other:"balance";[("balance()=@pre balance()
                        +amount")]
            guaranteed = true
            app_specific = false
            }
"Open"=>"Open"{
            event = "withdraw"
            parameter = "amount":"amount"
            condition = "(balance()-amount)" >= "0";[("amount=
                        o.balance();")]
            guaranteed = true
            app_specific = false
            }
"Open"=>"Open"{
```

```
            event = "balance"
            guaranteed = true
            app_specific = false
            }
"Open"=>"Inactive"{
            event = ""
            condition = "(getCurrentDate() - getLastActivityDate())" >=
                    "getMaxPeriod()";[("o.setLastActivityDate
                    (o.getCurrentDate()-o.getMaxPeriod());")]
            guaranteed = false
            app_specific = false
            }
"Open"=>"Frozen"{
            event = "freeze"
            guaranteed = false
            app_specific = false
            }
"Open"=>"Omega"{
            event = "close"
            guaranteed = true
            app_specific = false
            }
"Inactive"=>"Open"{
            event = "activate"
            guaranteed = false
            app_specific = false
            }
"Inactive"=>"Inactive"{
            event = "balance"
            guaranteed = false
            app_specific = false
            }
"Inactive"=>"Frozen"{
            event = "freeze"
            guaranteed = false
            app_specific = false
            }
"Inactive"=>"Omega"{
            event = "close"
            guaranteed = false
            app_specific = false
            }
"Frozen"=>"Open"{
            event = "unfreeze"
            condition = "balance" >= "0";
                    [("/* The following is a predicate assertion */

                        /** @assert(o.balance()>=0) */")]
            guaranteed = false
            app_specific = false
            }
"Frozen"=>"Inactive"{
            event = ""
            condition = "(getCurrentDate() - getLastActivityDate())" >=
                        "getMaxPeriod()";[("o.setLastActivityDate(
                         o.getCurrentDate()-o.getMaxPeriod());")]
            guaranteed = false
```

```
            app_specific = false
            }
"Frozen"=>"Frozen"{
            event = "balance"
            guaranteed = false
            app_specific = false
            }
"Frozen"=>"Omega"{
            event = "close"
            guaranteed = false
            app_specific = false
            }
```

D.1.2. Tool Outputs

The prototype version of the $FIST_2$ tool checked the given model description of the *NewAccount* FIC for the existence of one entry and one exit state and that all the states are reachable from the entry state. The results are provided to the user as shown in Figure D.2. In addition, the tool generated the following 22 Java test drivers from the *NewAccount* testable model using the all paths-state technique.

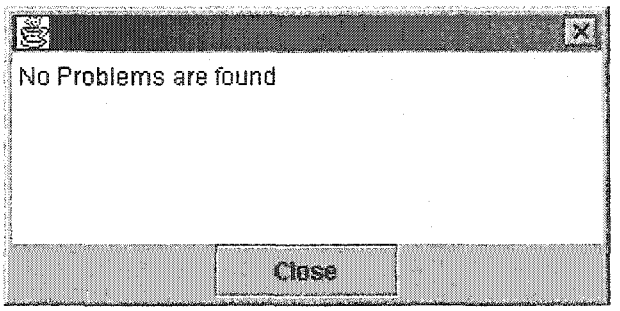

Figure D.2: Checking model result

```
public class TEST1_NewAccount{
    public TEST1_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
           NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
            o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Open, event:
            balance(), predicates: none */
        o.balance();

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
            o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}
```

```
public class TEST2_NewAccount{
    public TEST2_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
            NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
            o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Open, event:
           deposit(amount), predicates: amount=1.0 */
        amount=1;
        o.deposit(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
        o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST3_NewAccount{
    public TEST3_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
            NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
           o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Overdrawn, event:
           withdraw(amount), predicates: (balance - amount)<0 */
        amount=1+o.balance();
        o.withdraw(amount);

        /** @assert((o.balance()<0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Overdrawn, sink state: Overdrawn,
          event: balance(), predicates: none */
        o.balance();

        /** @assert((o.balance()<0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST4_NewAccount{
    public TEST4_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
            NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Overdrawn, event:
           withdraw(amount), predicates: (balance - amount)<0 */
        amount=1+o.balance();
        o.withdraw(amount);

        /** @assert((o.balance()<0) && ((o.getCurrentDate()-
         o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Overdrawn, sink state: Overdrawn,
         event: deposit(amount), predicates: (balance + amount)<0 */
        amount=o.balance();
```

```
        o.deposit(amount);

        /** @assert((o.balance()<0) && ((o.getCurrentDate()-
           o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST5_NewAccount{
    public TEST5_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
         NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
         o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Overdrawn, event:
         withdraw(amount), predicates: (balance - amount)<0 */
        amount=1+o.balance();
        o.withdraw(amount);

        /** @assert((o.balance()<0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Overdrawn, sink state: Open, event:
          deposit(amount), predicates: (balance + amount)>=0 */
        amount=1-o.balance();
        o.deposit(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST6_NewAccount{
    public TEST6_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Overdrawn, event:
          withdraw(amount), predicates: (balance - amount)<0 */
        amount=1+o.balance();
        o.withdraw(amount);

        /** @assert((o.balance()<0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Overdrawn, sink state: Omega, event:
          close(), predicates: none */
        o.close();

        /** @assert(((o.balance()<0.01)&&(o.balance()>-0.01))) */
    }
}

public class TEST7_NewAccount{
    public TEST7_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);
```

```
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Open, event:
          withdraw(amount), predicates: (balance()-amount)>=0 */
        amount=o.balance();
        o.withdraw(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST8_NewAccount{
    public TEST8_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
        /* Test transition: source state: Frozen, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());


        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Inactive, event:
          balance(), predicates: none */
        o.balance();

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST9_NewAccount{
    public TEST9_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
        /* Test transition: source state: Frozen, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());
```

```
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Open, event:
          activate(), predicates: none */
        o.activate();

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST10_NewAccount{
    public TEST10_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
        /* Test transition: source state: Frozen, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());


        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
    }
}

public class TEST11_NewAccount{
    public TEST11_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
        /* Test transition: source state: Frozen, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());
```

```
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Omega, event:
          close(), predicates: none */
        o.close();

        /** @assert(((o.balance()<0.01)&&(o.balance()>-0.01))) */
    }
}

public class TEST12_NewAccount{
    public TEST12_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
        /* Test transition: source state: Frozen, sink state: Open, event:
          unfreeze(), predicates: balance>=0 */
        /* The following is a predicate assertion */
        /** @assert(o.balance()>=0) */
        o.unfreeze();

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST13_NewAccount{
    public TEST13_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
        /* Test transition: source state: Frozen, sink state: Omega, event:
          close(), predicates: none */
        o.close();

        /** @assert(((o.balance()<0.01)&&(o.balance()>-0.01))) */
    }
}

public class TEST14_NewAccount{
    public TEST14_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);
```

```
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
        /* Test transition: source state: Frozen, sink state: Frozen, event:
          balance(), predicates: none */
        o.balance();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
    }
}

public class TEST15_NewAccount{
    public TEST15_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());


        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
        /* Test transition: source state: Frozen, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());


        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST16_NewAccount{
    public TEST16_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());
```

```
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
        /* Test transition: source state: Frozen, sink state: Open, event:
          unfreeze(), predicates: balance>=0 */
        /* The following is a predicate assertion */
        /** @assert(o.balance()>=0) */
        o.unfreeze();

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST17_NewAccount{
    public TEST17_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());


        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
        /* Test transition: source state: Frozen, sink state: Omega, event:
          close(), predicates: none */
        o.close();

        /** @assert(((o.balance()<0.01)&&(o.balance()>-0.01))) */
    }
}

public class TEST18_NewAccount{
    public TEST18_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());
```

```
        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Frozen, event:
          freeze(), predicates: none */
        o.freeze();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
        /* Test transition: source state: Frozen, sink state: Frozen, event:
          balance(), predicates: none */
        o.balance();

        /** @assert((o.balance()>=0) && (o.isFrozen())) */
    }
}

public class TEST19_NewAccount{
    public TEST19_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());


        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Open, event:
          activate(), predicates: none */
        o.activate();

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST20_NewAccount{
    public TEST20_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());


        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Omega, event:
          close(), predicates: none */
        o.close();
```

```
        /** @assert(((o.balance()<0.01)&&(o.balance()>-0.01))) */
    }
}

public class TEST21_NewAccount{
    public TEST21_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());


        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Inactive, event:
          balance(), predicates: none */
        o.balance();

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST22_NewAccount{
    public TEST22_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Omega, event:
          close(), predicates: none */
        o.close();

        /** @assert(((o.balance()<0.01)&&(o.balance()>-0.01))) */
    }
}
```

D.2. Application Development Stage

D.2.1. Tool Inputs

We have used the prototype version of the $FIST_2$ tool to test an implementation example of the FIC: *MyAccount* class. The *MyAccount* class can be implemented using the Hook Master tool. The *freeze* and *unfreeze* methods are not implemented in the *MyAccount* class and, therefore, the test drivers that cover them cannot be used to test the *MyAccount*

class. In addition, the application developer added a new method: *PrintStatement* that can be invoked at any model state. The prototype version of the tool is not integrated yet with the Hook Master tool to reflect the specification ignorance and addition performed when implementing the FIC. In our example, using the prototype version, the user deleted the Frozen state and the transitions associated with it and added the transitions for the specifications of the *PrintStatement* method (i.e., a self loop transition associated with the method call as an event was added to the open, inactive, and overdrawn states). This results in having the following model description written using the syntax listed in Appendix C.

```
"Alpha"{
        var_invariant = ""
        app_specific = false
        }

"Overdrawn"{
        var_invariant = "(o.balance()<0) && ((o.getCurrentDate()-
         o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())"
        app_specific = false
        }

"Open"{
        var_invariant = "(o.balance()>=0) && ((o.getCurrentDate()-
         o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())"
        app_specific = false
        }

"Inactive"{
        var_invariant = "(o.balance()>=0) && ((o.getCurrentDate()-
         o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())"
        app_specific = false
        }

"Omega"{
        var_invariant = "((o.balance()<0.01)&&(o.balance()>-0.01))"
        app_specific = false
        }

"Alpha"=>"Open"{
            event = "NewAccount"
            parameter = "amount":"float"
            condition = "amount" >= "0";[("float amount=1;")]
            guaranteed = true
            app_specific = false
            }
"Overdrawn"=>"Overdrawn"{
            event = "deposit"
            parameter = "amount":"float"
```

```
                condition = "(balance + amount)" < "0";[("amount=
                            o.balance();")]
                guaranteed = true
                app_specific = false
                }
"Overdrawn"=>"Overdrawn"{
                event = "balance"
                guaranteed = true
                app_specific = false
                }
"Overdrawn"=>"Open"{
                event = "deposit"
                parameter = "amount":"float"
                condition = "(balance + amount)" >= "0";[("amount=1-
                             o.balance();")]
                guaranteed = true
                app_specific = false
                }
"Overdrawn"=>"Omega"{
                event = "close"
                guaranteed = true
                app_specific = false
                }
"Open"=>"Overdrawn"{
                event = "withdraw"
                parameter = "amount":"float"
                condition = "(balance - amount)" < "0";[("amount=1+
                           o.balance();")]
                guaranteed = true
                app_specific = false
                }
"Open"=>"Open"{
                event = "deposit"
                parameter = "amount":"float"
                condition = "amount" = "1.0";[("amount=1;")]
                action = other:"balance";[("balance()=@pre balance()
                                +amount")]
                guaranteed = true
                app_specific = false
                }
"Open"=>"Open"{
                event = "withdraw"
                parameter = "amount":"amount"
                condition = "(balance()-amount)" >= "0";[("amount=
                            o.balance();")]
                guaranteed = true
                app_specific = false
                }
"Open"=>"Open"{
                event = "balance"
                guaranteed = true
                app_specific = false
                }
"Open"=>"Inactive"{
                event = ""
                condition = "(getCurrentDate() - getLastActivityDate())" >=
                        "getMaxPeriod()";[("o.setLastActivityDate
```

```
                    (o.getCurrentDate()-o.getMaxPeriod());")]
            guaranteed = false
            app_specific = false
            }
"Open"=>"Omega"{
            event = "close"
            guaranteed = true
            app_specific = false
            }
"Inactive"=>"Open"{
            event = "activate"
            guaranteed = false
            app_specific = false
            }
"Inactive"=>"Inactive"{
            event = "balance"
            guaranteed = false
            app_specific = false
            }
"Inactive"=>"Omega"{
            event = "close"
            guaranteed = false
            app_specific = false
            }
"Open"=>"Open"{
            event = "PrintStatement"
            guaranteed = true
            app_specific = true
            }
"Overdrawn"=>"Overdrawn"{
            event = "PrintStatement"
            guaranteed = true
            app_specific = true
            }
"Inactive"=>"Inactive"{
            event = "PrintStatement"
            guaranteed = false
            app_specific = true
            }
```

The generated test drivers given in Appendix D.1.2, from which the tool selected the non-broken test drivers, are input to the tool at the application development stage. The following code provides the Java implementation of the framework extended class (i.e., *Account*) and the implemented FIC (i.e., *MyAccount*). The *MyAccount* class is instrumented with Javadoc comments written in DbC for test case evaluation purposes. The instrumentation was done manually and will be automated when the tool is integrated with the Hook Master tool.

```
public class Account {
	Account(float amount) {
		this.Balance=amount;
	}
	float balance() {
		return Balance;
	}
	void withdraw(float amount) {
		Balance=balance()-amount;
	}
	void deposit(float amount) {
		Balance=balance()+amount;
	}
	float Balance;
}

public class MyAccount extends Account{
	/** @post ((balance()<amount+0.01)&&(balance()>amount-0.01))*/
	MyAccount(float amount) {
		super(amount);
		...
	}
	float balance() {
		return super.balance();
	}

	/** @post (balance()> ($pre(float,balance())-0.01-amount))&&(balance()<
	    ($pre(float,balance())+0.01-amount))*/
	void withdraw(float amount) {
		super.withdraw(amount);
	}

	/** @post (balance()> (amount+$pre(float,balance())-0.01))&&(balance()<
	    (amount+$pre(float,balance())+0.01))*/
	void deposit(float amount) {
		super.deposit(amount);
	}

	int getLastActivityDate() {
		return lastActivityDate;
	}
	/** @post(getLastActivityDate()==year)*/
	void setLastActivityDate(int year) {
		lastActivityDate=year;
	}
	/** @post $result == 5 */
	int getMaxPeriod() {
		return 5;
	}

	int getCurrentDate() {
		...
	}
	boolean isFrozen() {
		return frozen;
	}
	/** @pre (getCurrentDate()-getLastActivityDate())>=getMaxPeriod()*/
	/** @post (getLastActivityDate()-getCurrentDate())==0*/
	void activate() {
		lastActivityDate=getCurrentDate();
	}
	/* @post (balance()<0.01)&&(balance()>-0.01)*/
	void close() {
```

```
            withdraw(balance());
        }
        void PrintStatement() {
            ...
        }
        int lastActivityDate;
        boolean frozen;
}
```

D.2.2. Tool Outputs

1. Applicable test drivers

The tool found that the test drivers numbered 1-7 and 19-22 are applicable for testing the *MyAccount* class. In addition, the $FIST_2$ tool augmented three test cases to cover the new transitions associated with *PrintStatement* event. The following classes generated by the tool implement the augmented test cases. The added statements are bolded.

```
public class TEST23_NewAccount{
    public TEST1_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
           NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
            o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Open, event:
            PrintStatment(), predicates: none */
        o. PrintStatment();

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
            o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST24_NewAccount{
    public TEST3_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
            NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
           o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Overdrawn, event:
           withdraw(amount), predicates: (balance - amount)<0 */
        amount=1+o.balance();
        o.withdraw(amount);

        /** @assert((o.balance()<0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Overdrawn, sink state: Overdrawn,
          event: PrintStatment(), predicates: none */
        o.PrintStatment();
```

```
        /** @assert((o.balance()<0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}

public class TEST25_NewAccount{
    public TEST21_NewAccount(){
        /* Test transition: source state: Alpha, sink state: Open, event:
          NewAccount(amount), predicates: amount>=0 */
        float amount=1;
        NewAccount o = new NewAccount(amount);

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())<o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Open, sink state: Inactive, event:
          none, predicates: (getCurrentDate() -
          getLastActivityDate())>=getMaxPeriod() */
        o.setLastActivityDate(o.getCurrentDate()-o.getMaxPeriod());


        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
        /* Test transition: source state: Inactive, sink state: Inactive, event:
          PrintStatment(), predicates: none */
        o.PrintStatment() ();

        /** @assert((o.balance()>=0) && ((o.getCurrentDate()-
          o.getLastActivityDate())>=o.getMaxPeriod()) && !(o.isFrozen())) */
    }
}
```

2. Driver class

The $FIST_2$ tool detected the broken test drivers and generates the following driver to invoke the non-broken and augmented test drivers.

```
public class DRIVER_MyAccount{
     public static void main(String args[]){
         //invoking non-broken test drivers
         new TEST1_NewAccount();
         new TEST2_NewAccount();
         new TEST3_NewAccount();
         new TEST4_NewAccount();
         new TEST5_NewAccount();
         new TEST6_NewAccount();
         new TEST7_NewAccount();
         new TEST19_NewAccount();
         new TEST20_NewAccount();
         new TEST21_NewAccount();
         new TEST22_NewAccount();

         //invoking augmented test drivers
         new TEST23_NewAccount();
         new TEST24_NewAccount();
         new TEST25_NewAccount();
    }
}
```

3. FIC mapping class

The $FIST_2$ tool generated the following mapping class

```
public class NewAccount extends MyAccount{
        public NewAccount(float amount) {
             super(amount);
        }
        float balance() {
             return super.balance();
        }
        void withdraw(float amount) {
             super.withdraw(amount);
        }
        void deposit(float amount) {
             super.deposit(amount);
        }
        void activate() {
             super.activate();
        }
        void close() {
             super.close();
        }

        /* the following method is not generated automatically in the prototype
           version of the tool  */
        public boolean isFrozen() { return false; }
}
```

4. Testing Results

The $FIST_2$ tool compiled the non-broken test drivers, the driver class, the FIC mapping class, and the implemented FIC. Finally, the tool executed the driver class and used the Jcontract tool to produce the testing results shown in Figure D.3. The results are also stored in a log file as follows.

```
Jcontract: Version 1.5 -- Copyright (C) 2000-2002 ParaSoft
Jcontract: Environment:
    java.version = 1.3.1_01
    ...
    user.dir = C:\temp FIC\My Account\inputs to FIST thesis
Jcontract: Started on: 13/03/03 6:24 PM
...
Jcontract: Loaded instrumented class: TEST1_NewAccount
Jcontract: Loaded instrumented class: MyAccount
Jcontract: Loaded instrumented class: TEST2_NewAccount
Jcontract: Loaded instrumented class: TEST3_NewAccount
Jcontract: Loaded instrumented class: TEST4_NewAccount
Jcontract: Loaded instrumented class: TEST5_NewAccount
Jcontract: Loaded instrumented class: TEST6_NewAccount
Jcontract: Loaded instrumented class: TEST7_NewAccount
Jcontract: Loaded instrumented class: TEST19_NewAccount
Jcontract: Loaded instrumented class: TEST20_NewAccount
Jcontract: Loaded instrumented class: TEST21_NewAccount
Jcontract: Loaded instrumented class: TEST22_NewAccount
Jcontract: Loaded instrumented class: TEST23_NewAccount
Jcontract: Loaded instrumented class: TEST24_NewAccount
Jcontract: Loaded instrumented class: TEST25_NewAccount
```

```
Jcontract: Ended on: 13/03/03 6:24 PM

Jcontract: Runtime Statistics:
    Instrumented classes loaded: 15
    @pre          checks: 1
    @post         checks: 122
    @invariant    checks: 0
    @concurrency  checks: 0
    @assert       checks: 37
```

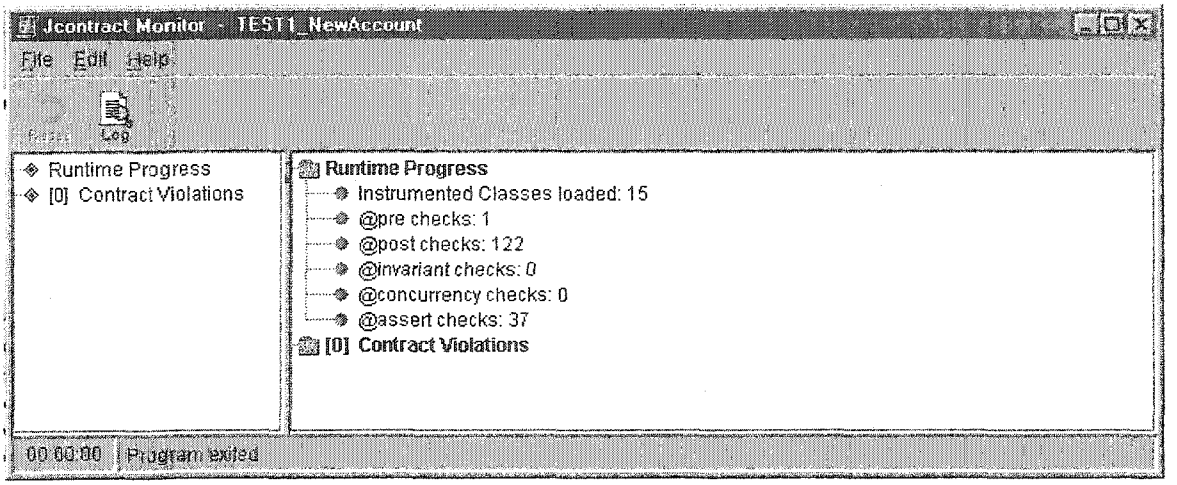

Figure D.3: The *MyAccount* class testing results produced using the Jcontract tool