2001

A study on test cases generation for object-oriented programs based on UML state diagram.

Xiaohong. Yang
University of Windsor

A Study on Test Cases Generation for Object-Oriented Programs Based on UML State Diagram

By

XIAOHONG YANG

A thesis
Submitted to the Faculty of Graduate Studies and Research
Through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor

Windsor, Ontario, Canada

April, 2001

Abstract

Software testing expenses are estimated to be between 20% and 50% of total development costs. Software testers need methodologies and tools to facilitate the testing portion of the development cycle. State-based testing is one of the most recommended techniques for testing object-oriented programs. Data flow testing is a code-based testing technique, which uses the data flow analysis in a program to guide the selection of test cases. Both state-based testing and data flow testing have their disadvantages. State-based testing does not analyze the program code, and thus could miss the detection of data members that do not define the states of the object class. Selecting data flow test cases from data members for testing classes is difficult and expensive. To overcome their weakness, a hybrid class test model is proposed, which contains both the information from specification about the state change of object instances of the Class Under Test (CUT) and the information from the source code about the definition and use of the data members in the CUT. With such an uniformed architecture, we can obtain automated tools to generate test cases for state-based testing and perform data flow testing at the same time. The combination of the two techniques is essential in improving our testing environment, and thus contributes to the enhancement of the reliability of software products. The proposed hybrid testing strategy can be used in both software design stage and software implementation stage. A Standard-based UML information exchange format--- XMI is used to describe UML design specification in the hybrid testing strategy, to bridge the software designer and software tester. No matter what kind of CASE tools designer use, as long as it is saved as XMI format, the testing tool can easily understand design specification from different design tools.

Keywords: class testing, hybrid testing strategy, Grey-box testing, data flow analysis, state-based testing, def-use pairs, finite state machine, state diagrams, XMI, UML

Acknowledgements

The work presented in this thesis could not have been possible without the support of many people.

First, I would like to express my sincere thanks and appreciation to my supervisor, Dr. Xiaojun Chen, for giving me the opportunity to do research in software-testing study. Her work ethics, advice, enthusiasm, and encouragement through the development of this work were the most important factors that helped me finished my M. Sc. program.

I would also like to thank my committee members, Dr. Diana Kao, Dr. Arunita Jaekel and Dr. Peter Tsin for spending their precious time to read this thesis and their comments, suggestion on this thesis work.

My special thanks go to the secretary of the School of Computer Science, Ms. Mary Mardegan, for her consistent helps. Thanks go to Li Fang , Xiubin, Hanmei, Wayne, for their valuable comments, interesting discussions, supportiveness, and kindness. I really appreciate all of their help.

I am deeply grateful to my husband, Ming Lei, for his love and support; and my parents, for their understanding and encouragement.

Table of Contents

List of Acronyms

CUT	Class Under Test
C-Use	Compute Use
CASE	Computer Aided Software Engineering
CSTD	Class State Transition Diagram
DOM	Document Object Model
DTD	Document Type Definition
Def-Use pair	Definition-Use pairs
EBNF	Extended Backus Naur Form
EFSM	Extended Finite State Machine
FSM	Finite State Machine
GUI	Graphic User Interface
HTCGS	Hybrid Test Case Generation System
JavaCC	Java Compiler Compiler
OMT	Object Management Team
OSD	Object State Diagram
P-Use	Predicate Uses
UML	Uniform Model Language
XMI	XML based Metadata Interchange
XML	eXtended Markup Language

List of Figures

List of Tables

Chapter 1

INTRODUCTION

This chapter gives an introduction to the work of this thesis. It includes its aim, motivation and objectives. It also discusses the features and advantages about framework of test cases generation based on hybrid testing strategy.

1.1 Aim

The aim of this thesis is to study and develop a new test cases generation method, which combines the state-based testing and data flow testing techniques. This technique is based on a test model called hybrid class test model, which contains both the information from specification about the state change of object instances of the Class Under Testing (CUT) and the information from the source code about the definition and use of the data members in the CUT. It belongs to a hybrid testing strategy. It generates basic test cases through navigating the class state transition diagram, and mapping each implemented class method's data occurrences into Hybrid Class Test Model. Then data anomaly detection can be applied. This method overcomes the shortage of state-based testing that may miss detection of code implementation errors, and significantly improves the performance of data flow testing. At the same time, it also reduces test case number and test cost for data anomaly detection. A strategy of test case generation is provided and a prototype of our object-oriented program test cases generation system is implemented.

1.2 Motivation

1.2.1 Why choose test cases generation as the main research focus in this thesis?

Expense on software testing is estimated to be between 20% and 50% of total development costs [Glass 1990]. Software testers need methodologies and tools to facilitate the testing portion of the development cycle. In software testing, test case generation is the most difficult, expensive and tedious. It can easily take several months

of hard work. Because test cases generation is labor intensive and expensive, automating the process could significantly reduce the costs of software development.

1.2.2 What challenges does software testing have if tested programs are object-oriented?

Object-oriented programming is easier to maintain through better data encapsulation and the reuse of classes, but object-oriented techniques have made software testing harder due to encapsulation, inheritance, data binding, overriding/overloading and polymorphism.

1.2.3 Why use UML state diagram as one of the main testing resources that generate test cases?

There is an increasing need for effective testing of software in design stage. UML (Uniform Model Language) provides a powerful mechanism for describing software and it is widely used in software design. Advantage of using software description languages such as UML is that they provide a convenient way for generating test cases. Although UML is not completely formalized, certain aspects of a language, in particular the UML State chart diagram, are precise enough to be utilized for test generation.

Software design descriptions represent a significant opportunity for testing because they precisely describe how the software functions behave in a form that can easily be manipulated by automated means. Design notations can be used as a basis for output checking, significantly reduce one of the major costs of testing. The process of generating test cases from design will help the test engineer discover problems with the design itself. If this step is done early, the problems can be eliminated early. It saves time and resources. Generating test cases during design also allows testing activities to be shifted to an earlier part of the development process, which allows for more effective planning and utilization of resources.

1.2.4 Respective disadvantages of State-based testing and Data-flow testing

Some testing techniques (e.g. state-based testing and data flow testing) used to test conventional programs have been adapted for testing object-oriented programs. State-based testing is one of the most popular techniques for testing object-oriented classes. State-based testing only observes changes of state values and focuses on state dependent behaviors of objects. The state values are combined values from its data members at a particular point in time [Tuner and Robson, 1993]. This means that state-based testing only involves data members, which have an effect on behaviors of an object. Some data members in a class may exist for other purposes and do not define object states. Those unexamined data members in state-based testing need to be detected by some other techniques in order to ensure the quality of implemented classes. State-based testing only examines state change and behavior rather than internal logic, such that data faults may be missed. State-based testing does not examine program code, and thus could miss the detection of data members that do not define the states of an object class.

Data flow testing is expensive when it is used for object-oriented program testing. Selecting data flow test cases from data members for testing classes at the intra-class level is difficult and expensive. Weyuker [1984, 1990] proposed a technique to generate test cases that covers all du-paths (definition-use path). The method reaches an exponential number of test cases in the worst-case scenario. If d is the number of two way decisions in the program, then in the worst case *all du-paths* require $O(2^d)$ of time to generate test cases. Meantime some of the test cases may be infeasible, as the selection is based on the definition-use pairs. Furthermore, the data flow test cases cannot be completely generated if any data flow anomalies exist in the class, because the anomalies may break the def-use pairs that are the basis of selecting data flow test cases.

Each testing strategy has its own purpose. State-based testing transforms a class behavior model into test cases. Data flow testing generates test cases based on the patterns of definitions and uses of the data members in a class. State-based testing does not detect data members that do not define states. However, this can be examined using data flow testing.

1.2.5 Test case generation can be paralleled with system design and implementation phases in software development life cycle

There are some existing software development life cycle models.

The Waterfall development model progresses from the analysis phase to the design phase, through to the coding and finally the testing phase. In this model, there is necessity to revisit earlier phases of development. This is because the implications of a decision in a phase could not have been foreseen until it was worked through in later phases. In addition, if faults are not found until the testing phase, then we need to revisit previous life cycle phase in order to fix the faults. This may cause the product being delivered late. It increases costs.

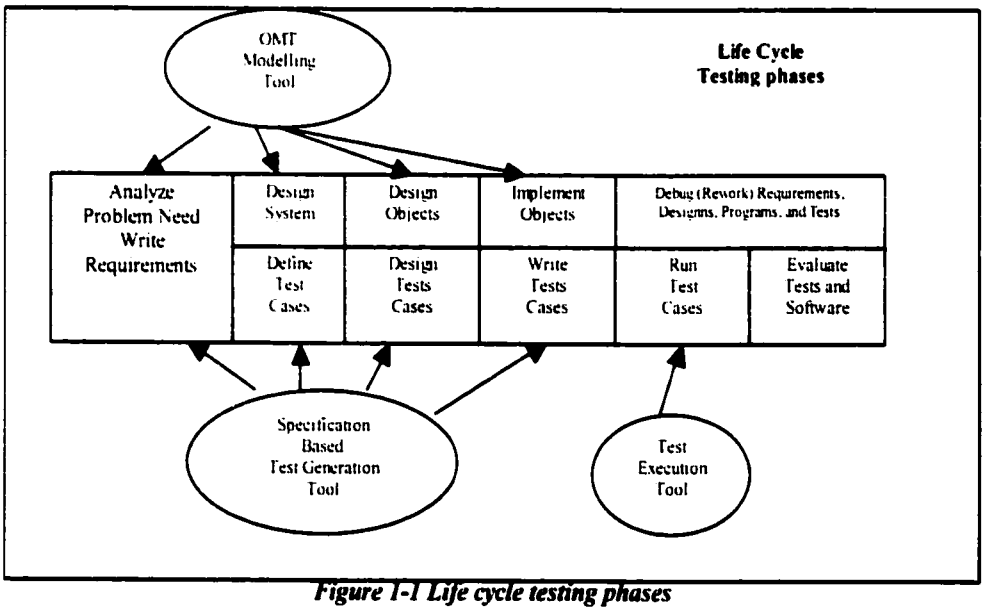


Figure 1-1 Life cycle testing phases

James Rumbaugh and his colleagues developed the Object Modeling Technique (OMT) [Rumbaugh etc., 1991]. The testing phase of the life cycle follows the implementation phase. During the testing phase, defining test cases, designing test cases, writing tests cases, and then running test cases are performed sequentially. Finally, the software is evaluated. In this model, the testing tasks are also performed after an object is implemented. The tasks of defining, designing, and writing test cases cannot be performed in parallel with the tasks of defining, designing and implementing objects.

In our solution, test cases should be developed alongside requirements, design and coding (see Figure 1-1). Testing should not be envisaged as a distinct phase of the life cycle but as an integral part to the design and build process. Testing is put in each stage in our hybrid testing method. After software design stage, specification-based testing (mostly State-based testing) is needed. When implementation coding finishes, specification-based testing and code-based testing need to be done to detect errors. The ability of detecting existing data anomaly shows that code-based testing is a necessary testing step.

1.2.6 Why XMI is used as information exchange format in the hybrid testing format

Automatic testing requires a standard information exchange format, which allows software designer and software tester (could be testing software) to exchange design specification (now popular based on UML).

In software development life cycle, when designers finish their design, the design result is saved as some kind of file. In order to automate testing, testing tool should read and understand the designer's design file. Unfortunately, the gap between designer and tester still exists. Most researchers just manually input the design specifications. Pioneer researcher Jeff Offutt propose to parse some specific design document, like Rational Rose' mdl file to get the semantic meanings of the specifications (Jeff Offutt, 1999). But this is a vender dependent solution. A standard-based, vender independent UML information exchange format is needed to bridge the software designer and tester, otherwise, automatic testing can not be language independent.

1.3 Objectives

Both state-based testing and data flow testing have their disadvantages. To overcome the weakness, adapt and improve the existing testing techniques, Grey-box object-oriented class testing technique is presented. This kind of Grey-box object-oriented class testing technique is a hybrid method, based on state based testing and data flow testing.

This method is not a simple way to combine state based testing and data flow testing. It maps methods of the implemented CUT onto the transitions of the state model, in order to generate sequences of implemented methods. Based on the sequences, intra-class data flow test cases can be selected using the conventional def-use path selection technique (also used by Harrold and Rothermel [1994]).

It is very expensive to generate test cases only from source code based on data flow testing. Applying the hybrid method, the data flow testing technique can be used in detecting data members to find data faults that occur within those data members. These data faults may not be examined in state-based testing. The proposed data flow testing technique is used on intra-class testing. It concerns tracing the flow of data members among methods in the class, rather than local variables within an individual function. Moreover, all definitions and uses are associated with class methods rather than statement blocks in a function. The number of d-u (definition-use) paths is reduced. Hybrid testing strategy keeps the advantage of data flow testing and significantly reduces the testing cost.

The hybrid testing strategy can be used in software design stage and software implementation stage. In design stage, testers can start design test cases and do simulation testing. After the coding phase, data flow testing can be applied based on the state information. It does not need to reengineer source code to get state information. Expense on test case generation is reduced significantly.

A Standard-based UML information exchange format--- XMI (XML based Metadata Interchange) is used in my research work, to bridge the software designer and software tester. No matter what kind of CASE tools designer use, as long as it is saved as XMI format, the testing tool can easily understand design specification from different design tools.

1.4 Thesis Structure

This thesis consists of seven chapters. Chapter 2 discusses research background. It includes kinds of testing strategy, state-based testing and data flowing testing, and evaluates advantages and shortcoming of each method. Chapter 3 describes the grey-box hybrid class testing strategy, and chapter 4 presents the design and implementation of a prototype of test cases generation system for Object-Oriented Program. Chapter 5 shows the results of the evaluation of our approach. Chapter 6 describes the comparison of this work with related work done so far by other researchers. Finally, Chapter 7 provides conclusions and recommendations for future work.

Chapter Two

RESEARCH BACKGROUND

This chapter evaluates the techniques of state-based testing and data flow testing for testing object-oriented classes. It also introduces the UML state diagram, and gives a brief discussion about UML information exchange standard---XMI.

2.1 State-based Testing

2.1.1 What is state-based testing?

State-based testing is to examine the interactions within an object by monitoring the changes of states, and errors are detected by testing states of an object.

State-based testing emphasizes on testing the state change of an object due to possible method calls in each state.

Usually, state-based testing exercises an implementation to see whether it produces the correct response for different use case and message sequences. A state machine model can represent both control responsibilities and the class interface that implements them. Test suits that are effective at finding control bugs can be generated from these models.

2.1.2 State-based testing approaches

In state-based testing, as in all types of dynamic class testing, the object of a class is tested rather than the class itself, since the latter is not executable.

State-based testing is often realized according to given specification about the behavior of the instances of the Class Under Testing (CUT).

With state-based testing methods, we can generate test cases by traversing test trees [Chow, 1978], spanning trees [Kung et al., 1994, 1996] or state-transition trees [Binder, 1995]. Test cases can be derived from Finite State Machines (FSMs) [Chow, 1978] or extended state models [Kung et al., 1993, 1994, 1996; Binder, 1995]. Some state-based testing methods manually inspect test results using the state models.

2.1.3 State-based testing for object-oriented programming

In an object-oriented language, a class contains both methods and data members. Private members are encapsulated within the class and are invisible from outside. An object encapsulates its states (i.e. the combinations of the values of data members), and only the methods of the object can be referred to or modify its data members. Therefore, the behavior of an object may be controlled by its states and message sequences. Additionally, classes are usually designed to accept certain sequences of messages. They may respond differently for a particular set of encapsulated values. Based on these characteristics of object classes, state-based testing is proposed to observe the values stored in the state of objects and focus on the state dependent behavior of the objects. The benefit of state-based testing lies in its ability to examine the interactions within an object by monitoring the changes of states.

State-based testing is one of the most recommended techniques for testing object-oriented programs. This technique was originally proposed by Turner and Robson [1993]. Their approach is based on FSMs, and determines input states and output states for each method from the design of the class. The changing states rely on the values that are changed by the transitions. Feasible method sequences, which are generated along with the class specification (e.g. state transition diagram), can be used to check whether the object-oriented program follows the methods sequence patterns required by specification. The testing processes are: (1) for every state s, to create an object and set it to the state, (2) to invoke the operation f with the parameter x and chosen test data, and (3) the resulting state s' is compared with the state predicated by the transition function.

2.1.4 State transition diagram to a state transition tree

Adopting state-based testing, Binder in [1995] transcribes the state transition diagram of an *Account* class to a state transition tree, and the tree is used to generate test cases. This state transition tree, the *test-tree* derived using the *Object Test Model* in [Tse and Xu, 1995, 1996] and the *reachability tree* are similar to the spanning trees used in [Kung et al., 1996, 1993; Chow 1978]. A test case is generated after traversing each full or partial branch in the tree.

Here we give an often-used example *stack* to express the state transition tree. Assume that the bounded *stack* class can only store five units of data and has *push* and *pop* methods. The state spaces of a *stack* object consists of six states, from the empty state, to one containing one unit, to another containing two units, etc., to the full state.

In practice, an *n* array size of *queue* has *n+1* state spaces (state value is from *0* to *n*). From a modelling perspective, the state space diagram is not acceptable since every change of state value results in a *"new"* state. Therefore, the state space must be divided into a more manageable and meaningful substate size. For this example, the behaviour of *stack* suggests three states: *empty*, *loaded*, and *full*. The methods *push* and *pop* are modelled as events. Sending ***pop*** to an *open* stack results in *empty* stack when the popped item was the only item in the stack. Sending push to a *loaded* stack results in a *full* stack only if the stack contains one less item than the maximum allowed before the push. In all other cases, push and pop result in the loaded state.

The *stack's* state transition diagram in Figure 2-1 is used in an object design for describing its state changes and behaviours. In Figure 2-2, the stack's state transition diagram is transcribed to a state transition tree

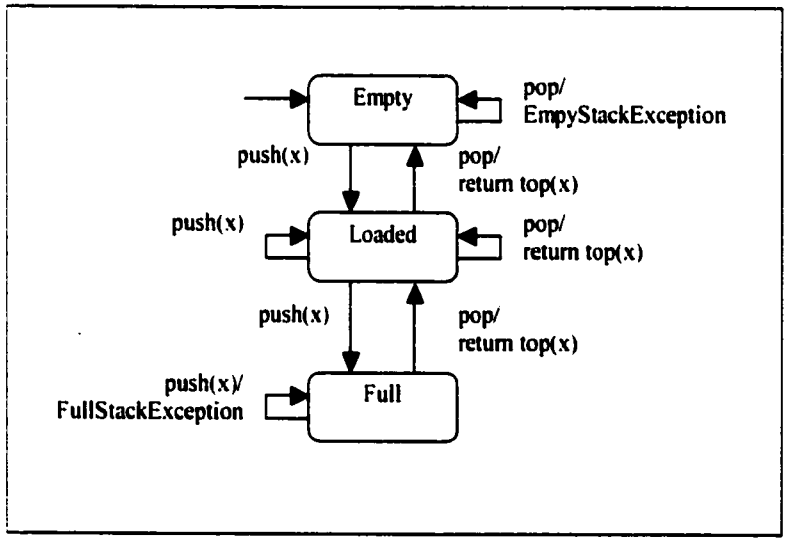


Figure 2-1 *State transition diagram of the stack class*

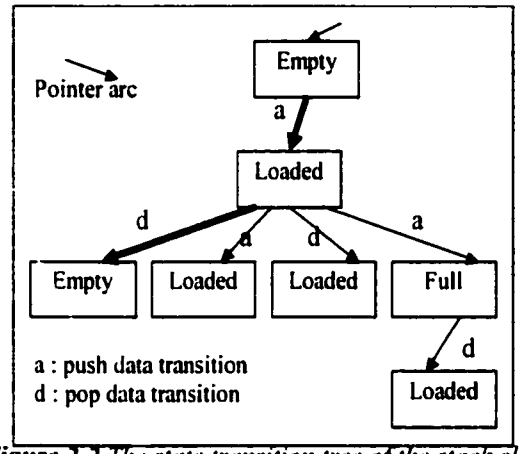


Figure 2-2 *The state transition tree of the stack class*

2.1.5 Test Case Generation using state transition tree

The method of producing test cases from state transition sequences has been discussed in [Chow, 1978; Binder, 1995 and Kung et al., 1996]. The first step in generating test cases for the class under test is to prepare a *state transition tree* (or spanning tree).

Generally, the initial state in the state transition diagram becomes the root node in the tree, and each transition out of the initial state is a branch to a node that represents the resultant state of the transition. Therefore, the next possible states of the initial state in the diagram are the child nodes of the root node on the tree. This is repeated for each subsequent state to build the tree until (1) the node already appears on the previous level or (2) the node corresponds to a final state in the state transition diagram [Chow, 1978; Binder, 1995].

Each path from the root in the tree comprises a possible transition sequence from the beginning state to the end state in the state transition diagram. A path also shows a sequence of test cases to detect whether or not the state changes correctly. The state transition tree derived from the state transition diagram of the *stack* class is given in Figure 2-2. A sequence or a single test case is obtained by tracing each full or partial path in the tree. For example, a sequential test case <*Empty, push, Loaded, pop, Empty*> is produced by following the bold branches in Figure 2-2.

2.2 Data flowing Testing

2.2.1 what is data-flow testing?

Data flow testing is a code-based testing technique, which uses the data flow analysis in a program to guide the selection of test cases [Harrold and Rothermel, 1994; Frankl and Weyuker, 1988]. For object-oriented programs, it can be used in method_level testing as well as intra_class testing.

2.2.2 Data-flow testing approaches

Since the 1980's, numerous data flow-testing approaches have been discussed in the literature [Harrold and Rothermel, 1994]. Most of these techniques are based on data flow analysis and require test data (cases) to exercise definition-use paths.

2.2.3 Data-flow analysis in data-flow testing

The advent of data flow analysis used in compiler optimization has also affected the strategies of software testing. In conventional program testing, data flow analysis of the program is a mechanism for selecting execution paths, which should be tested.

Data flow analysis is the examination of the use of variables in a program, i.e. the sequences of actions on the variables. An improper use of data is a data anomaly [Chan and Chen, 1987]. The important types of data flow anomalies stated in [Beizer, 1990; Fosdick and Osterweil, 1976] consist of: (1) defining a variable twice with no intervening use, (2) using a variable that is killed, and (3) releasing variables that are defined but not used.

Data flow analysis examines where/how variables are **defined** with values and where/how the values of the variables are **used**. A test path is formed from definitions to uses in a program.

A definition-use pair (**def-use** pair) is an ordered pair (**d**, **u**), where a statement called **d** contains a definition of a variable *v*, which is used in a statement **u** in a program.

Uses of a variable are further divided into two classes, as either computation uses (**c-use**) or predicate uses (**p-use**) [Rapps and Weyuker, 1985]. A c-use occurs when the value of a variable is used in a computation or output statement, and a p-use occurs when the value is used in a condition (predicate) statement. For instance, the *if (x > 0) {x = y + 10;}* statement contains a p-use of *x* and a c-use of *y*, followed by a def of *x*. The data flow testing criteria can be found in [Harrold and Rothermel, 1994; Frankl and Weyuker, 1988].

A variable is killed (***k***) when its value is released or it contains no known value. "*Killed*" (or undefined) is that the *instance* of a variable is killed [Beizer, 1990]. The anomalies that are often used are listed in Table 2-1

Table 2-1 *The actions of data flow anomaly*

Actions on a variable	Anomaly
dd	A define action is followed by a define action. Probably a harmless anomaly but strange.
ku	A kill (undefined) action is followed by a use (reference) action. A harmful anomaly, the value is released before reference.
dk	A define action is followed by a kill action. Why was the variable defined but not used? Probably an anomaly.
d-	A variable was defined without usage. Probably an anomaly, but this could be a global definition.
-u	A variable is used without definition. Probably an anomaly, but the variable may have been previously defined.

A major characteristic of classes is the interaction between data members and methods. This interaction is represented as definitions and uses of data members within methods. Data flow testing criteria are based on the patterns of definitions and uses of the program variables. They test the individual data definition-use relationships. Therefore, data flow testing can be applied in detecting those data members, which are not examined in state-based testing, in order to find data faults that occur within those data members.

When executing a data flow test case, the test case is said to exercise a def-use path if the path is traversed.

Many researchers [Fosdick and Osterweil, 1976] have used data flow analysis to detect programming faults known as "data flow anomalies". When the pattern of use of variables is abnormal, there is an anomaly in the data flow, such as misspelling and confusion of variable names, omission of statements, incorrect parameter usage, and so on. For instance, variables having the "referencing killed variable anomaly" are usually due to a misspelling.

2.2.4 Control Flow Graphs in data-flow analysis

A control flow graph is used to represent the control structure of a program, and the graph is defined in Harrold [1993] states:

> *"A control flow graph is a directed graph where each node in the graph represents a basic block and each edge represents the flow of control between basic blocks."*

In a control flow graph, there is an edge on the graph from node D_i to node D_j if D_j immediately follows D_i in some execution sequence. This also indicates that D_j is a successor of D_i, and D_i is a predecessor of D_j. Edges representing conditional transfer of control are labelled "T" (true) or "F" (false) on the graph, other edges are unlabeled. To represent entry into the program (function) and exit from it, a start node (named "s") and a terminal node (named "t") are added to a control flow graph.

In data-flow analysis, it can be used to generate data member occurrences by traveling the def_use paths in a control flow graph.

2.3 State transition diagrams

A state transition diagram is a graphic representation of a state machine. Nodes represent states. Arrows (directed edges) represent transitions. The annotations on the edges represent events and actions.

A *class state transition diagram* only contains states, transitions and events, although state transition diagrams for classes may contain states, events, conditions, transitions and actions [Booch, 1994]. An action is an operation and is associated with an event. "*A state can be defined in terms of a condition; conversely, being in a state is a condition*" [Rumbaugh et al., 1991]. For deriving sequences of test messages and parsing test results, only the states and events are used.

Definition: A *class state transition diagram* (CSTD) of a class C can be modeled as 4-tuple (S, E, T, s), where

- S is a finite set of states, i.e., S = {s | s = (*def*)} where *def* is a predicate on state attributes.
- E is a set of events, which are receivable for the objects of C.
- T is a finite set of transitions, i.e., T = $\{(s_i, t, s_j) \mid s_i, s_j \in S \text{ and } t \in T\}$.
- s is a virtual state representing the period before an object is created.

Using the *stack* class specification as an example, the process of constructing a class state transition diagram is demonstrated in the following subsections. The above example stack's CSTD corresponding to its dynamic behavior is given in Figure 2-1.

States

The state of an object is determined by the values of its attributes. Sets of values are grouped together into a state according to properties that affect the behavior of the object [Rumbaugh et al, 1991]. This also means that the states of an object can be formed by the sets of state values. The object of a class may have a large range of state spaces (values). For modeling the state space in CSTD, they must be partitioned to a more manageable *substate* size and each of them shares the properties of interest. For example, the array size of a stack is *n*. There may be from *0* to *n* elements on the stack as indicated by the *top* attribute. The state space for the stack can be divided by *top=0*, *0<top<n* and *top=n*. These three substates are considered and depicted in a diagram rather that *n* states.

Events

An event causes the state of an object to change [Martin and Odell, 1992; Booch, 1994], and it serves as a marker for a particular time when the state change occurs. Events are sent from client objects and replies may be expected. When an object receives an event, the state changing to the next state depends on the current state and the event. This implies that the current state is a precondition, when it is true, the event triggers some actions, and then the state is changed. The specification may explicitly state what events the object can receive.

State Transitions

A state transition represents the fact that an event's occurrence causes the state of an object to change. Each transition is drawn as a direct arc to connect two states. A state may be a source state and target state of a transition (i.e. it has a state transition to itself), and a state may have many different unique state transitions. These are labeled with the events that cause the transitions, and point to the next states.

2.4 XMI for UML information exchange

2.4.1 What is XMI?

XMI is an acronym for "XML Metadata Interchange". XMI is a standard from OMG (Object Management Group). XMI solves the general interchange problem by creating an open interchange format from a domain representation using UML.

The main purpose of XMI is to support the exchange of metadata (structured data describing object based models for information systems) between modeling tools based on the OMG UML (an object based modeling language). XMI is an OMG architecture, which is aimed at supporting interoperation of information systems within diverse and distributed installations.

XMI uses XML so that XMI data exchanges can take advantage of the extensive technology development surrounding XML and the Web.

'XMI' is made up of:

- Rules for generating XML Document Type Definitions (DTDs) from metamodels (these metamodels are descriptions of classes of metadata; these classes of metadata may include the specifications of the metamodels themselves, to as many levels as required)
- Rules for generating XML Documents from metadata
- Design guidelines for XMI-related DTDs and XML data
- Actual DTDs supporting UML

XMI has been developed in response to an OMG requirement for a Stream-based Model Interchange Format, in the specific context of UML.

As there are different vendor based CASE (Computer aided Software Engineering) tools, all these tools support UML. Their serialization code is different. For instance, like Rational Rose's mdl file and select TogetherSoft Enterprise's enp file cannot exchange. Meanwhile in software development life cycle, software designer and software tester can be different people. How can testing tool understand the designer's idea is a problem.

XMI can fill this gap. Software designer provides an XMI file for their UML design specification. Testing tool can easily parse this XMI file and get the UML exchangeable information, generate test cases and plan software testing automatically.

2.4.2 XMI file format

XMI's XML document production process is defined as a set of production rules. When these rules are applied to a model or model fragment, the result is an XML document. The inverse of these rules can be applied to an XML document to reconstruct the model or model fragment.
ENBF Rules Representation the XML produced by XMI is represented here in Extended Backus Naur Form (EBNF).

The following are the production rules:

1. <Document> ::= <2:XMI>

```
2.  <XMI> ::=          "<xmi:XMI"
                       <2a:Namespaces>
                       ( <3:Header> )?
                       ( <6:Content> )?
                       (<4:Differences> )?
                       ( <5:Extensions> )?
                       "</xmi:XMI>"
 ......
3. <Header> ::=        "<xmi:header>"
 ...
                        "</xmi:header>"
3a. <Documentation> ::= "<xmi:documentation>" //text//
 ...
                       "</xmi:exporterVersion>" )*
                      "</xmi:documentation>"
3b. <Model> ::=       ( "<xmi:model name=" //name//
  ...
                       "</xmi:model>" )*
3c. <Metamodel> ::=   ( "<xmi:metamodel name=" //name//
                       "version=" //version//
                       <2e:Link>? ">" //text//
                        "</xmi:metamodel>" )*
3d. <Metametamodel> ::= ( "<xmi:metametamodel name=" //name//
```

The XMI EBNF representation can reference in XMI specification document. I omit the rest long document about XMI file EBNF format.

2.4.3 XMI DTD architecture

Every XMI DTD contains the elements generated from information, plus a fixed set of element declarations that may be used by all XMI documents. These fixed elements provide a default set of data types and the document structure, starting with the top-level XMI element. Each XMI document contains one or more elements called XMI that serves as a top-level container for the information to be transferred. The XMI element contains the following structural elements:

- Header, which contains version declarations and optional documentation regarding the transfer
- Content, which contains the core information that is to be transferred
- Difference, which specify the differences between two XMI documents to be transferred.
- Extensions, which allows the transfer of private tool information beyond that already present in a DTD.

Generating element declarations provides an architectural consistency for each element in several important areas: Object identity, extensibility, navigation, and linking. This consistency lets tools know how to traverse any XMI document or DTD in a regular manner to find the information needed.

- Object identity is standardized through the xmi.element.att.macro (called an XML entity), which declares markers for optionally declaring unique identity via uuids, plus optional shorthand labels and local document ids.
- Extensibility is standardized by the optional xmi.extension element which allows nested extensions in addition to those present in the extensions section.
- Navigation is standardized by the regular patterns from which DTDs are generated.
- Linking is standardized through the xmi.link.att.macro.

2.5Terminology

2.5.1 Test Cases

Test cases are sets of input data (or series of test messages) that demonstrate whether the software program performs as expected [Lorenz, 1993]. They can be designed using black-box, white-box or Grey-box strategies [Beizer, 1995].

Object-oriented class testing is used to detect errors in the implementation of the object states and/or the methods. Therefore, the test cases of object-oriented programs focus on the state and the methods involved with the state changes. Most existing work in class testing has selected sequences of messages to test for errors. One reason why test cases are made of sequences of messages is that the execution paths of each message are determined by the states of the object.

2.5.2 Test Messages

In an object-oriented program, objects mainly interact via messages, which may be used to request a corresponding service or to provide data (return the value of a parameter). Firesmith [1993] states: "*The sender and the receiver of a message have numerous, and different responsibilities. Testers should test to determine whether any of these responsibilities have failed to be met.*" In Turner and Robson [1993], test results are evaluated by sending sequences of test messages to an object under test. This is a common approach for object testing.

2.5.3 Test Oracles

Test result checking is one of the most important considerations in automated testing [Beizer, 1990]. A test oracle is a means by which the behavior of a software system or program can be verified. Turner [1995] defines: "*A test oracle is a person or program that is able to determine whether the output or results generated by a test case represent the expected values.*"

Chapter Three

Hybrid testing strategy for Object-Oriented Programs Testing

In this chapter, a hybrid testing strategy for Object-Oriented programs testing is proposed which is based on state based testing and data flow testing. This method is not a simple way to combine state based testing and data flow testing. It maps methods of the implemented Class Under Test (CUT) onto the transitions of the state model, in order to generate sequences of implemented methods. Based on this sequence, intra-class data flow test cases can be selected using the conventional def-use path selection technique.

Applying the hybrid method, the data flow testing technique can be used in detecting data members to find data faults that occur within those data members. These data faults may not be examined in state-based testing. On the other hand, the data flow testing technique in the hybrid method is concerned with tracing the flow of data members among methods in the class, rather than local variables within an individual function. The *hybrid class test model* keeps the advantage of data flow testing and significant reduces the testing cost.

3.1 A priority bounded queue class example for hybrid testing strategy

For explaining the hybrid testing strategy, a queue class is adopted. Assume the following specification expresses the requirements of the printer queue class for a large system:

> *"The printer queue class has limited space to contain five units of data (jobs). Each print job has been assigned a priority. A higher priority job is processed before any job of lower priority, and two jobs with the same priority are processed according to the order in which they were added to the queue object.*
> *A created printer queue object provides insert job, remove job, and check size services. As a job is inserted, it will be stored at the rear of the queue object, if the object has a space. The front job in the queue is removed, if the object receives a removal message and the queue is nonempty. The check size service returns the current job items in the queue object."*

The UML state diagram of queue class is in Figure3-1, and its implementation source code is shown in Figure 3-2.

In the class there are four methods, plus a constructor and a destructor. Moreover, a *sort()* is declared as a private function, which cannot be called from the outside of the *queue* class. When a *queue* object is declared, the constructor *queue()* causes the object to be at the *Empty* (initial) state, and the data members *f, r, count and Q[]* array are initialized. The *Q[Size]* is declared for storing job data. The *f* and *r* data members are two array indices, used to indicate the position from/in, which the next data will be deleted/added. Moreover, the *count* data member is used to calculate the amount of data added to an object of the *queue*. The *add()* method is executed when a non-full *queue* object receives an "*insertion*" message. If a "removal" message is sent to an object and its state is non-empty, the *del_data()* method can be performed. The statements of both the *add()* and *del_data()* functions show that the *queue* class is implemented as a circular model. Rather than shifting items left when a job data is deleted, the queue elements are logically arranged in a circle. The data member *count* maintains a record of the number of jobs in the *queue*, and if *count* equals *Size*, the *queue* object is full. The *is_empty()* and *sizes()* methods look at the current status of the current data items in an object. The *add()* method is executed when a non-full *queue* object receives an "*insertion*" message. If a "removal" message is sent to an object and its state is non-empty, the *del_data()* method can be performed.

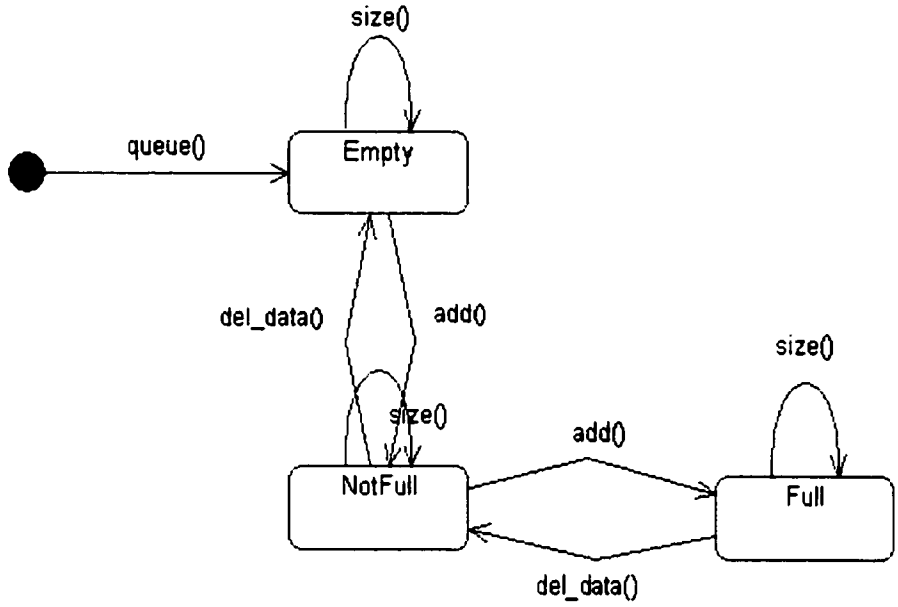


Figure 3-1 the UML state diagram of the queue

```
const int Size = 5;
class queue{
protected: char Q[Size];   // bounded array
           int  f, r;      // Front/Rear index of queue
           int  count;     // a counter of the array
           void sort();    // decreasing order
public:    queue(void);    // default constructor
           void is_empty(void);
           int  add(char);// add data to the queue
           char del_data();//delete data from the queue
           void sizes(void);// get the size of the queue
           ~queue(void);  // destructor
};
queue::queue(void){
      r = -1;      f = 0;
      count = 0;
      for (int i = 0; i < Size; i++){
        Q[i] = ' ';}
}
int queue::add(char data){
      if (count == Size){
        cout << "Not room for adding new data to stack\n";
        return (0);}
      r++;
      if(r==Size){
        r=0;}
      Q[r] = data;
      count++;
      sort();      // Enable this class to be a priority queue
      return (1);
}
```

```
char queue::del_data(){
      char data;
      if (count == 0){
        cout << "can't delete data from an empty
                stack\n";
        return('0');}
      data = Q[f];
      f++;
      if(f==Size){
        f=0;}
      count--;
      return (data);
}
void queue::is_empty(){
      if (count == 0)
        cout<<"The queue is empty\n";
      else
        cout<<"The queue is non-empty\n";
}
void queue::sizes(){
      cout << "the size of the queue :" << count
                << "\n";
      return;
}
queue::sort(){
      // Bubble sort or others can be adopted here.
}
queue::~queue(){
      cout << "\n releasing the queue object \n";
}
```

***Figure 3-2** Implemented C++ code of the* queue *class*

3.2 Framework of hybrid testing strategy

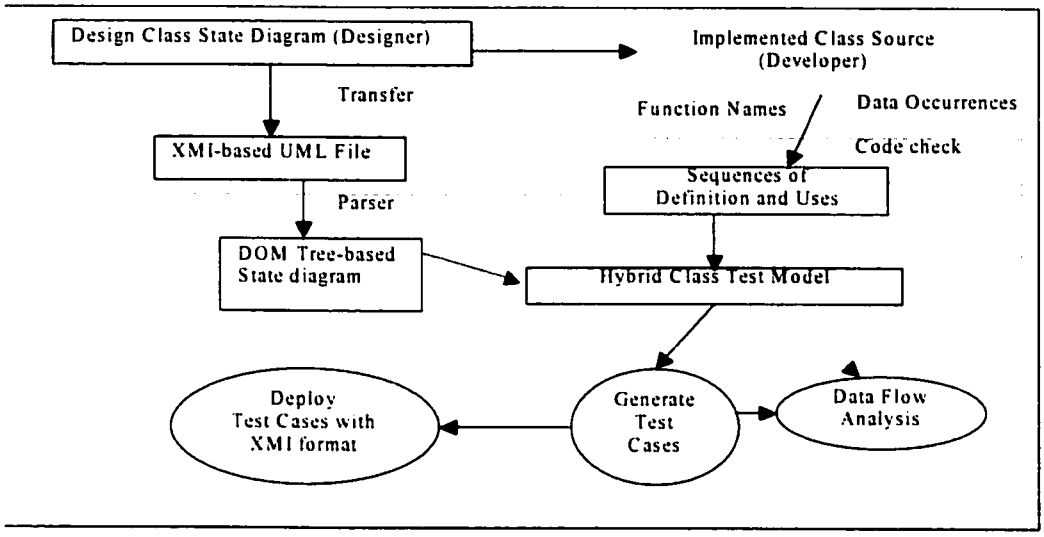


***Figure 3-3** Hybrid Class Test Architecture*

The hybrid class test architecture can be seen in Figure3-3.

At first, software designer designs the software according to software requirements. This design specification follows UML design notation, and then it is saved as a XMI format file. The design specification is sent to implementation team and testing team.

Second, the testing tool parses the XMI format design specification, builds a DOM tree based class state diagram. No matter source code has been finished or not, the testing tool can build a Hybrid Class Test Model from design specification. Then pure State-based cases can be generated through Hybrid Class Test Model. This kind of test cases can be used to detect the design errors.

Third, when source code has been implemented, the implementation team sends the source code to testing team. From the CUT definition, we can retrieve information about the definition and use of data members in order to perform data flow testing/analysis. With the help of the class state diagram, the testing software put source code into def-use information generator, generate the def-use pair information, and then map the information for each given method in a given state. These def/use sequences are then added to the transitions on the class state diagram. The obtained new diagram is the hybrid class test model. Using this model, the testing tool generates State-based test cases and data flow test cases. Through analyzing state-based test cases, the testing tool checks state related design and implementation errors. Through analyzing the data flow

definition-use information, the testing tool detects the data anomaly in code implementation, and finds possible solution to remove the data anomaly.

From above describing, the hybrid testing strategy can be used in different software development stage. In software design stage, we can use this strategy to do test case generation, although code has not been implemented. The design errors could be found. After the code has been implemented, the design specification and source code both are available. Hybrid testing strategy can generate both state-based and data flow test cases. Through running these test cases, we can find design errors and implementation errors.

Here, the test case generation framework provides a flexible mechanism to handle state-based testing and data-flow testing advantages, and overcome shortcomings of each. Meanwhile it also provides Object-Oriented programs automatic testing solution. Further, the design specification can be transferred between design software (CASE tool) and testing software with XMI format. Test cases (method sequences, can be shown in sequential diagram) also can be transformed in XMI format. So test cases can be deployed and executed in a distributed system.

3.3 Extracting UML design specification by parsing XMI

In our research, assume that software designer has already finished the software design and provided the UML design specification with XMI format.

XMI is a kind of XML file. If testing software would like to use the information, it need parse this file according to UML DTD.

A XMI parser is built to do this job. This XMI parser is a DOM (Document Object Model) parser. It builds a DOM tree, and then translates DOM tree to State transition diagram. At last, we get related information.

After analyzing object behavior, the following information needs to be identified:

- The states an object can have;
- The state transitions that can occur;
- The events that can occur; and
- The operations that can take place as an event occurs.

- There are a finite number of states and transitions;
- The behavior of the object can be defined with the states;
- The transitions between the states depict the state change;
- All states can be reached from the initial state; and each state has a unique name.

3.4 *Static analysis by scanning the data flow paths to get sequences of definition and use*

In a class definition, the methods of an object can define, use or define-use its data members. Therefore, global definitions and uses of each data member in methods can be extracted by static analysis through scanning the data flow paths.

The global definitions and uses of data members within each method should be computed first to enable the detection of data flow anomalies of data members within the sequences of methods. While computing global definitions and uses of data members, the static data flow analysis can be applied to detect whether any data anomalies occur on the local variables or data members within the function.

As we mentioned in Chapter 2, a directed flow graph, also called a control flow graph, contains the definition and use of data variables. It can be used to generate the data member occurrences. The data occurrences are shown within programs and facilitate the computation of def-use pairs.

In detecting intra-class data anomalies and selecting intra-class data flow test cases, only the data members in the public methods or in the private methods are concerned.

The global definition and use of data members within each function are considered in our research. *This means in each function, we just record the inter-method data member definition and user information. If the data member's definition-use pairs are already matched inside a function, we will omit these kinds of data occurrences.*

For example, to simplify the presentation of each method of *queue*, each code statement is a unit in which the definition and/or use of data members can occur. Each statement number in the *add()* function corresponds to a node number in the control flow graph, see Figure3-4. Nodes 1 and 3 represent a conditional transfer of control and each one has two labeled edges emerging from it. For example, node 3 has two successor nodes, 4 and 5.

```
int que::add(char data){
1.  if (count == Size){
      cout << "Not room for new \n";
      return (0);
    }
2.  r++;
3.  if(r==Size){
4.    r=0;}
5.  Q[r] = data;
6.  count++;
7.  return (1);
}
```

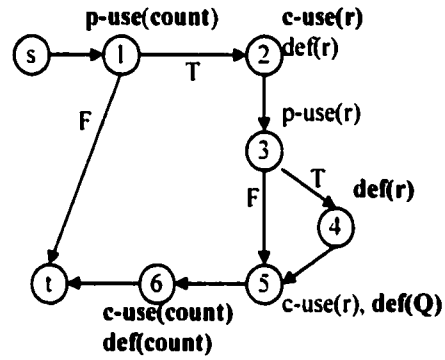


Figure 3-4 *The C++ code add() member function on the left and its directed flow graph on the right. Nodes in the graph represent statements in the function; start and terminate nodes are added to aid analysis.*

From Figure3-4, the **p-use(*count*)**, **c-use(*r*)** and **c-use(count)** at nodes 1, 2 and 6 are concerned as to judge, whether the *count* and *r* data members used in this function have been properly defined in the preceding functions. Moreover, it is necessary to examine whether the defined *r*, *Q* and *count* data members, i.e. **def(*r*)**, **def(*Q*)** and **def(*count*)**, can be used in the succeeding functions.

In Figure 3-4, the bold definitions and uses of data members should be computed in addition to the non-bold data members. Here we have the definition-use pairs: dpu(*r*, 2)={3}, dcu(r, 2)={5} and dcu(*r*, 4)={5} which have formed sub paths within the *add()* function. Whether the data member *r* (defined at statements 2 or 4) or the data members *Q* and *count* (defined at statement 5 and 6 respectively) will be used in the succeeding functions, will need to be examined.

The global definition and use of data members within each function are revealed in its implemented transition, such as:

***Empty**, add(data), sort(), **NotFull**, (0<count<5), {p-use(count), c-use(r), def(r), def(Q), c-use(count), def(count)}*

To clear and simplify the representation, we use this format:

Empty → NotFull (add(data), A)

where A:[(use, count), (use, r), (def, r), (def, Q), (use, count), (def, count)]

3.5 Using state diagram and global data occurrences to construct Hybrid Class Test Model

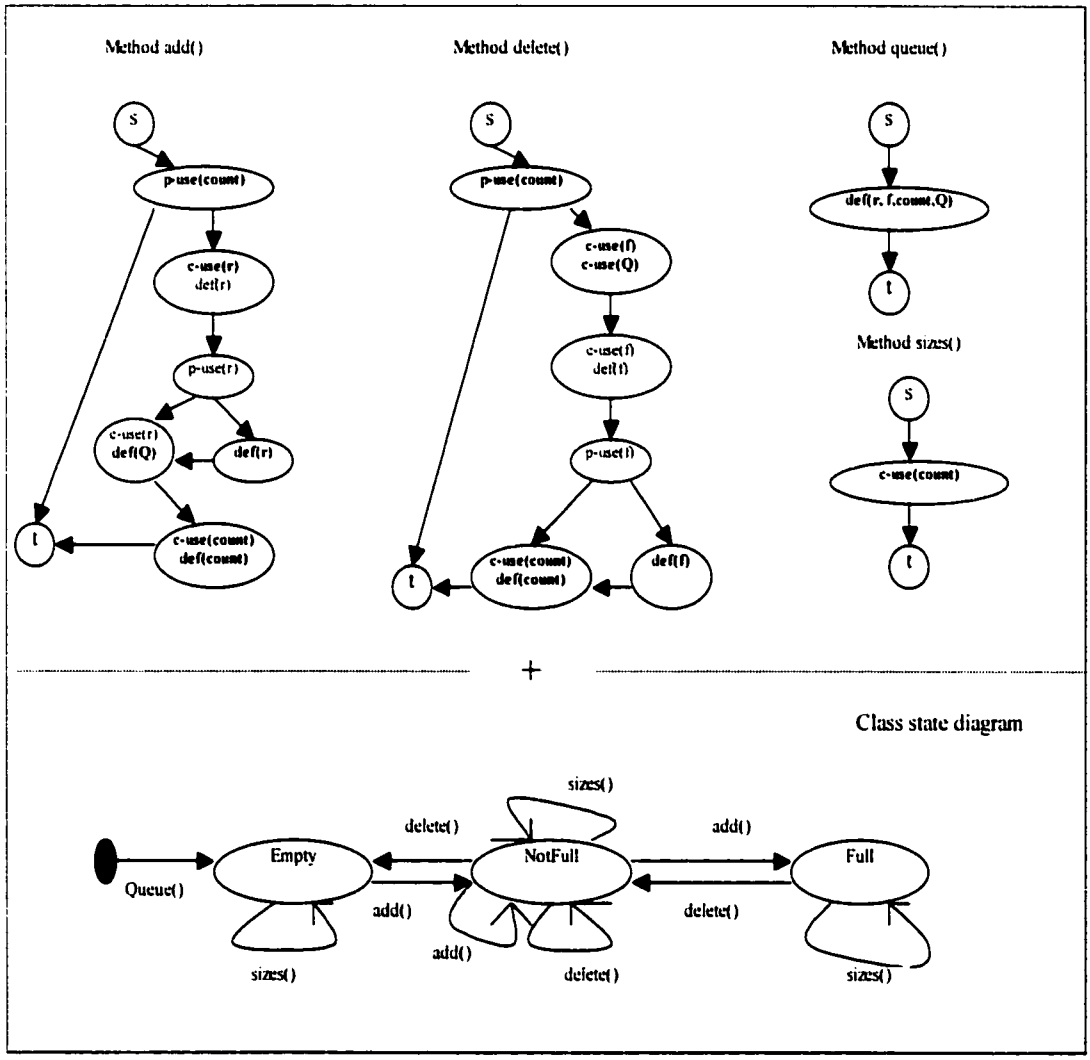
In hybrid testing, we can build a Hybrid Class Test Model from two fields. One field is state diagram, which is constructed from XMI file. A state diagram models those states that are reachable from any given state by messages sent to methods. The other is global data occurrence of each method, which is extracted by static analysis scanning the data flow paths (expressed by control flow graph). A control flow graph models how states are computed. It does not show how the method that the control flow graph represents is related to other methods.

The Hybrid Class Test Model that integrates method control flow graphs is constructed by joining state transition diagram and control flow graphs (Figure 3-5). It provides a model of all intraclass control flow paths (Figure 3-6). Such a model can provide the missing linkages among class methods. So a Hybrid Class Test Model is composed of two parts: one representing data members and methods, the other representing states, transitions, and data occurrences in methods.

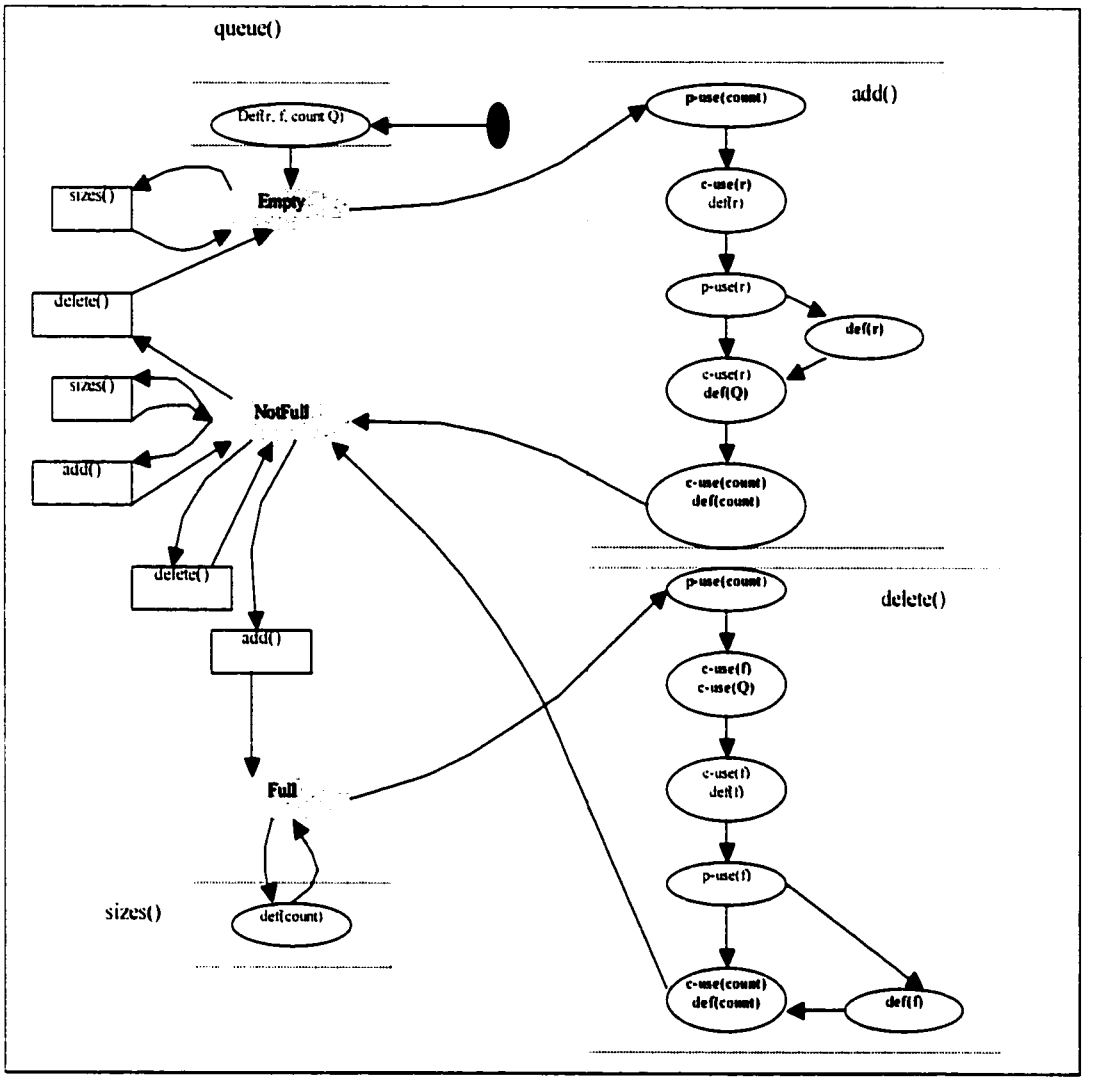
The basic construction of the Hybrid Class Test Model that integrates control flow graphs is illustrated with class *queue*. Class *queue* has four main methods: add(), delete(), queue() and sizes(). Figure 3-5 shows two parts of class queue. The first part shows the four method control flow graphs. The second part illustrates the state model for class *queue*. Because state is the result of method activation, we can substitute method graphs

into state transitions. The control flow graphs are imbedded in the state model to suggest how each state is computed (Figure 3-6).

The Hybrid Class Test Model that integrates control flow graphs is a class scope, code-based model for path analysis. It shows how paths in each method may be followed by paths in other methods, thereby supporting CUT path analysis.



Figur3-5 *Method control flow graph and class state diagram of class queue*



Figur3-6 *State diagram that integrates control flow graph of* ***queue*** *class*

Here we give the **definition** of **Hybrid Class Test Model.**

We assume that the behavior of an object of the CUT is specified in a class state diagram. A class state diagram shows the possible states of an object instance of the CUT, the possible calls of all public methods defined in the CUT in each state, and the state change of the object due to the method calls. Precisely, a class state diagram is a finite state machine (S, V, F, M, →) where

- S is a finite set of states. It denotes the set of possible states of the instances of the CUT.
- V is a finite set of data members. It contains the data members that we are interested in the CUT.
- For each state s in S, F(s) returns the values of each data member in the CUT.
- M is a finite set of public method names appeared in the CUT.
- $\rightarrow \subseteq S \times M \times S$ is a ternary relation that denotes the set of transitions. $(s1, m, s2) \in \rightarrow$, also written as $s1 \xrightarrow{m} s2$, denotes the possibility of state change from s1 to s2 trigged by method *m*.

As an example, Figure 3-7 shows a Hybrid Class Test Model of class *queue*.

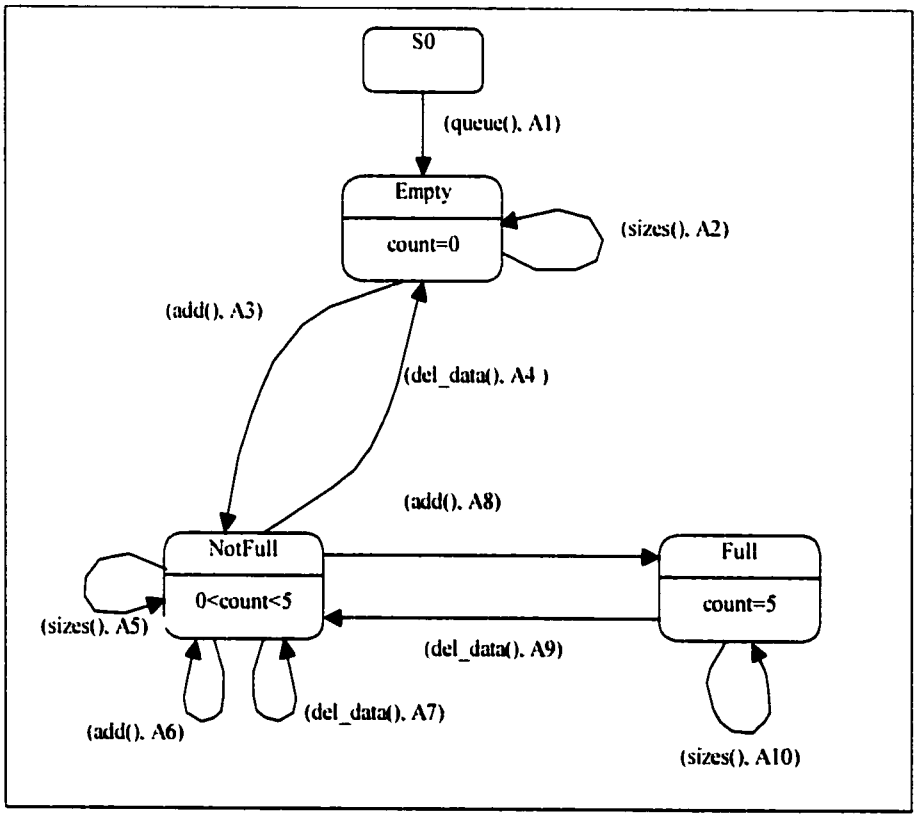


***Figure 3-7** Hybrid Class Test Model of the **queue** Class*

where

- S={S0, Empty, NotFull, Full}
- V = {count}

- F is defined as follows:

  F(S0) undefined

  F(Empty) (count)=0

  0<F(NotFull) (count) <5

  F(Full)(count)=5

- M={ queue( ), sizes( ), add( ), delet_data(void)}
- → contains

  S0 $\xrightarrow{\text{queue( )}}$ Empty

  Empty $\xrightarrow{\text{sizes( )}}$ Empty

  Empty $\xrightarrow{\text{add( )}}$ NotFull

  NotFull $\xrightarrow{\text{del-data( )}}$ Empty

  NotFull $\xrightarrow{\text{sizes( )}}$ NotFull

  NotFull $\xrightarrow{\text{add( )}}$ NotFull

  NotFull $\xrightarrow{\text{del-data( )}}$ NotFull

  NotFull $\xrightarrow{\text{add( )}}$ Full

  Full $\xrightarrow{\text{del-data( )}}$ NotFull

  Full $\xrightarrow{\text{sizes( )}}$ Full

Using white-box testing techniques, we can parse the source code (Figure 3-2) to retrieve information about the definition and use of the data members in the CUT.

A1: [(def,count), (def,r), (def, f), (def, Q)]

A2: [(use, count)]

A3: [(use, count), (use, r), (def, r), (def, Q), (use, count), (def, count)]

A4: [(use, count), (use, f), (use, Q), (def, f), (use, count), (def, count)]

A5: [(use, count)]

A6: [(use, count), (use, r), (def, r), (def, Q), (use, count), (def, count)]

A7: [(use, count), (use, f), (use, Q), (def, f), (use, count), (def, count)]

A8: [(use, count), (use, r), (def, r), (def, Q), (use, count), (def, count)]

A9: [(use, count), (use, f), (use, Q), (def, f), (use, count), (def, count)]

A10: [(use, count)]

3.6 From Hybrid Class Test Model to generate state-based test cases and data member occurrences sequences

The method of producing test cases from state transition sequences has been discussed in [Chow, 1978; Binder, 1995 and Kung et al., 1996].

In Chow's work [Chow, 1978], a testing tree is derived from Finite State Machines (FSMs). Test cases are manually generated by traversing test trees [Chow, 1978]

In Binder's work [Binder, 1995], he transcribes the state transition diagram of an *Account* class to a state transition tree, and the tree is used to generate test cases.

In Kung's work [Kung et al., 1996], a program-based state testing technique, has been proposed. The testing steps of the technique are:

1. to produce an object state diagram (OSD) from any C++ program using a reverse engineering tool, and then
2. to analyse the object state behaviours and generate test cases for testing object state interaction using a composite object state testing tool.

In conclusion of the above three people's work, the first step in generating test cases for the class under test is to prepare a *transition tree* (or spanning tree). The transition trees can be derived from the state transition diagram in Binder's work, the finite state machines in Chow's work or the object state diagrams in Kung et al.'s work. The nodes of the transition tree represent the states of the diagram. The edges of the tree represent transitions between the states.

We applied the method referred in section 2.1.5. The initial state in the Hybrid Class Test Model becomes the root node in the tree, and each transition out of the initial state is a branch to a node that represents the resultant state of the transition. Therefore, the next possible states of the initial state in the diagram are the child nodes of the root node on the tree. This is repeated for each subsequent state to build the tree until (1) the node

already appears on the previous level or (2) the node corresponds to a final state in the Hybrid Class Test Model.

Each path from the root in the tree comprises a possible transition sequence from the beginning state to the end state in the Hybrid Class Test Model. A path also shows a sequence of test cases to detect whether or not the state changes correctly. The transition tree derived from the Hybrid Class Test Model of the *queue* class is given in Figure 3-8. A sequence or a single test case is obtained by tracing each full or partial path in the tree. For example, a sequential test case <*Empty*, **Add**, *NotFull*, **Delete**, *Empty*> is produced by following the bold branches in Figure 3-8.

By navigating the transition tree (Figure 3-8), we can get the test cases:

- *Test case1*: <Empty, add, NotFull, Delete, Empty>

Data Occurrences Sequence:

(def, r), (def, f), (def, count), (def, Q), (use, count), (use, r), (use, count), (def, r), (def, Q), (def, count), (use, count), (use, f), (use, Q), (use, count), (def, f), (def, count)

- *Test case2*: <Empty, add, NotFull, add, Full, Delete, NotFull>

Data Occurrences Sequence:

(def, r), (def, f), (def, count), (def, Q), (use, count), (use, r), (use, count), (def, r), (def, Q), (def, count), (use, count), (use, r), (use, count), (def, r), (def, Q), (def, count), (use, count), (use, r), (use, Q), (use, count), (def, f), (def, count)

- Test Case 3: < Empty, add, NotFull, add, NotFull>

Data Occurrences Sequence:

(def, r), (def, f), (def, count), (def, Q), (use, count), (use, r), (use, count), (def, r), (def, Q), (def, count), (use, count), (use, f), (use, Q), (use, count), (def, f), (def, count)

- Test Case 4: < Empty, add, NotFull, delete, NotFull>

Data Occurrences Sequence:

(def, r), (def, f), (def, count), (def, Q), (use, count), (use, r), (use, count), (def, r), (def, Q), (def, count), (use, count), (use, r), (use, count), (def, r), (def, Q), (def, count)

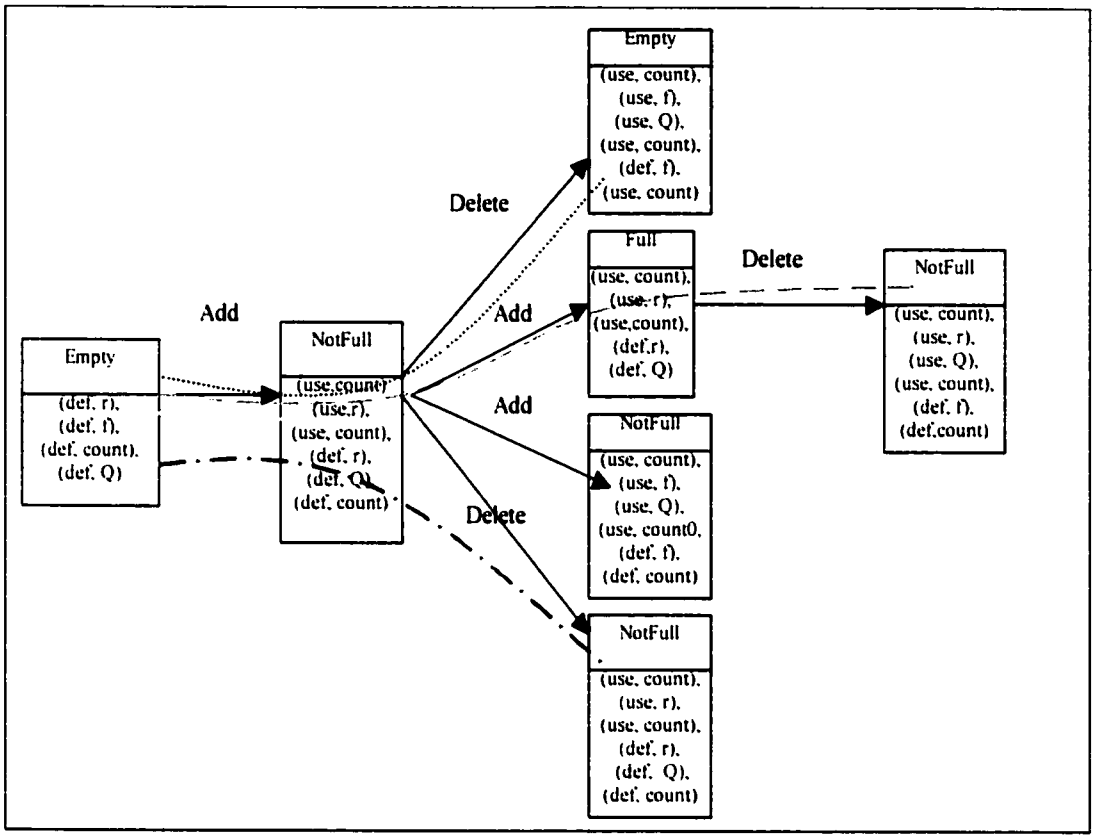


Figure 3-8 *The transition tree of the queue class from Hybrid class testing model*

3.7 Data flow anomalies detection technique within sequences of methods

By checking code, data flow anomalies can be found. It is similar to the cost that the compiler using the information known at compiling time can detect some data flow anomalies. However, the method sequences are unknown until run-time when the sequences of messages are sent from the client objects in an arbitrary order. For detecting data anomalies of data members within the feasible sequences of methods, the approach given in chapter 2 is adopted in this hybrid testing technique. Therefore, the usage of every data member can be stored in the transition tree in order to generate sequences of actions on each data member and facilitate data anomaly detection.

The transition paths in the Hybrid Class Test Model show all feasible sequence behaviors of the required object. Based on the transition paths, the data anomaly detection steps of this hybrid class test method are:

(1) Revealing the global definitions and uses of data members within functions.

In object-oriented classes, the methods of an object can define, use or define-use its data members. Therefore, global definitions and uses of each data member in methods can be extracted by scanning the data flow paths.

(2) Transforming the state transition diagram to Hybrid Class Test Model.

After getting all global definitions and uses of data members within functions, mapping these def-use pairs to form the Hybrid Class Test Model.

(3) Generating sequences of occurrences of data members from Hybrid Class Test Model.

Each node in the Hybrid Class Test Model (see Figure 3-7) contains a function name and the data occurrences within the function. Using transition tree concept, the sequences of def-use information can be generated by traversing the paths in the transition tree.

(4) Detecting data anomalies of data members within the generated sequences.

In general, an anomaly on a data member in a sequence of data occurrences occurs if one or more of the ***dd***, ***ku*** and ***dk*** data anomalies exists in the sequence.

More detail can be seen in our case study.

3.8 Transferring test cases to XMI format to deploy each testing software

The design specification is transferred between design tool (CASE tool) and testing tool with XMI format. Testing tool can easily parse the vendor-independent format design information, and build its testing model. Further, after the test cases are generated, testing software distributes test cases to respective test driver with XMI format in distributed system. Each test case can be run in different environment, different platform with different program languages. Testing tool can deploy test cases with XMI format to a distributed testing environment. Distributed processes coordinate their activities with XMI format. In respective distributed process, the test driver transfers test cases to detailed test program in right language, which fit for its environment and platform. This will significantly improve flexibility of distributed testing system.

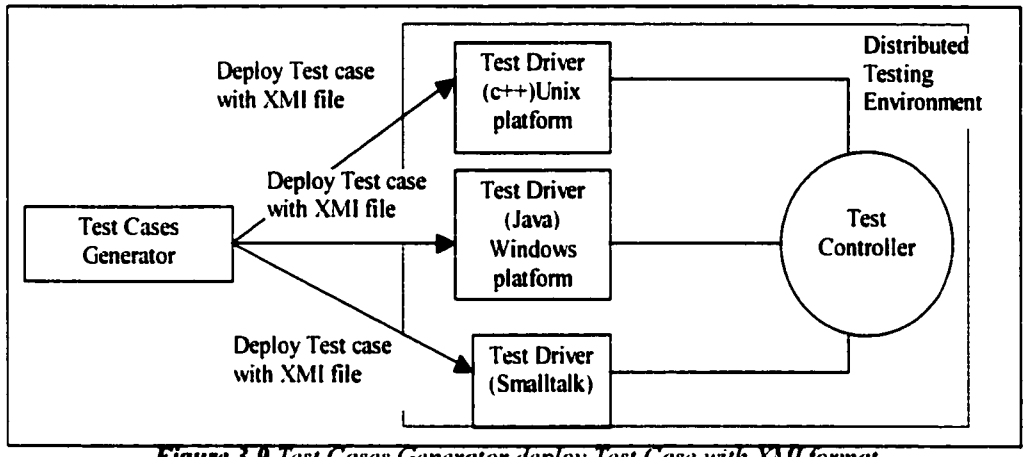


***Figure 3-9** Test Cases Generator deploy Test Case with XMI format in distributed testing environment*

3.9 Case study

A coin box component of a vending machine is presented in [Kung et al., 1996] as an example to demonstrate their testing approach. The example called class *CCoinBox* is adopted as the study case.

3.9.1 Case Description

The coin box component in [Kung et al., 1996] has very simple function. Only quarters (25-cent coins) are receivable and vending is allowed when two quarters are received. The functions of the component consist of *adding a quarter*, *returning the current quarters*, *resetting it at the initial state* and *vending*. The C++ source code of the *CCoinBox* class illustrated in the paper is shown in Figure 3-10.

There is an error in the implemented *CCoinBox* class. Kung et al. detects the error via state testing. They report:

> *"... that there is an error in the implementation. But the error is not obvious. We argue that the error cannot be easily detected by functional testing and/or structural testing of the methods since: ..., the error was due to interactions involving more than one method through an object state."*

```
class CCoinBox {
    unsigned totalQtrs;                          //total quarter collected
    unsigned curQtrs;                            //current quarters collected
    unsigned allowVend;                          // 1 = vending is allowed
public:
    CCoinBox() {Reset();}
    void AddQtr();                               //add a quarter
    void ReturnQtrs() {curQtrs = 0;}             //return current quarters
    unsigned isAllowVend() {return allowVend;}
    void Reset() {totalQtrs = 0; allowVend = 0; curQtrs = 0;}
    void Vend();                                 //if allowed, update totalQtrs and curQtrs
};
void CCoinBox :: AddQtr() {
    curQtrs = curQtrs + 1;                       //add a quarter
    if (curQtrs > 1)                             //if more than one quarter is collected
      allowVend = 1;                             //then set allowVend
}
void CCoinBox :: Vend() {
    if (isAllowVend()) {                         //if allowVend
      totalQtrs = totalQtrs + curQtrs;           //update totalQtrs,
      curQtrs = 0;                               //curQtrs, and
      allowVend = 0;                             //allowVend,
    }                                            //else no action
}
```

Figure 3-10 *The C++ source code of class CCoinBox* [adopted from Kung et al., 1996]

3.9.2 Applying Hybrid testing to Class CcoinBox

The test approach is achieved by

(1) Using state-based testing to generate test cases and inspect test results, and

(2) Following the data flow analysis approach to detect data anomalies within the sequences of methods.

An appropriate State Diagram to describe the behavior of the coin box component yielded at detailed design level is required. Assume that it is similar to the one presented in Figure 3-11.

➢ The State Diagram and Hybrid Class Test Model

UML state diagrams are used in our approach for generating test cases. The UML State Diagram, Figure 3-11 describes the behavior of the *CCoinBox* class.

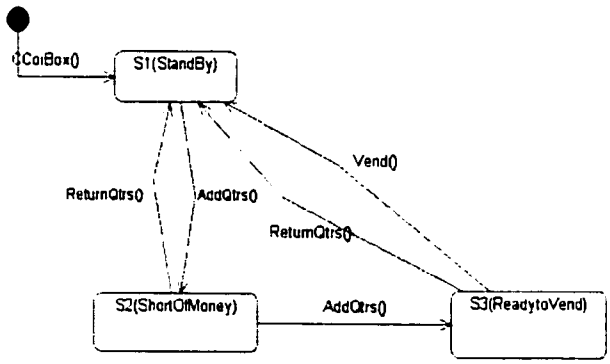


Figure3-11 *State Diagram of CoinBox from software designer*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE XMI SYSTEM "uml.dtd">
<XMI xmi.version="1.0">
  <XMI.header>
    <XMI.metamodel xmi.name="UML" xmi.version="1.3"/>
  </XMI.header>
  <XMI.content>
    <Model_Management.Model xmi.id="xmi.1" xmi.uuid="10-6-3-14--
1d8e66ee:e42aa66809:-8000">
     ...
     <Behavioral_Elements.State_Machines.StateMachine.context>
                <Foundation.Core.Class xmi.idref="xmi.2"/>
     </Behavioral_Elements.State_Machines.StateMachine.context>
     <Behavioral_Elements.State_Machines.StateMachine.top>
        <Behavioral_Elements.State_Machines.CompositeState xmi.id="xmi.4">
<Foundation.Core.ModelElement.name>state_machine_top</Foundation.Core.ModelElement
name>
     <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
     <Behavioral_Elements.State_Machines.State.stateMachine>
       <Behavioral_Elements.State_Machines.StateMachine xmi.idref="xmi.3"/>
     </Behavioral_Elements.State_Machines.State.stateMachine>
     <Behavioral_Elements.State_Machines.CompositeState.subvertex>
        <Behavioral_Elements.State_Machines.Pseudostate xmi.id="xmi.5"
xmi.uuid="10-6-3-14--1d8e66ee:e42aa66809:-7ffc">
     <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
..

       <Behavioral_Elements.State_Machines.Transition xmi.idref="xmi.11"/>
          </Behavioral_Elements.State_Machines.Event.transition>
        </Behavioral_Elements.State_Machines.SignalEvent>
      </Foundation.Core.Namespace.ownedElement>
    </Model_Management.Model>
  </XMI.content>
</XMI>
```

***Figure 3-12** The XMI fragment of class CcoinBox UML design specification*

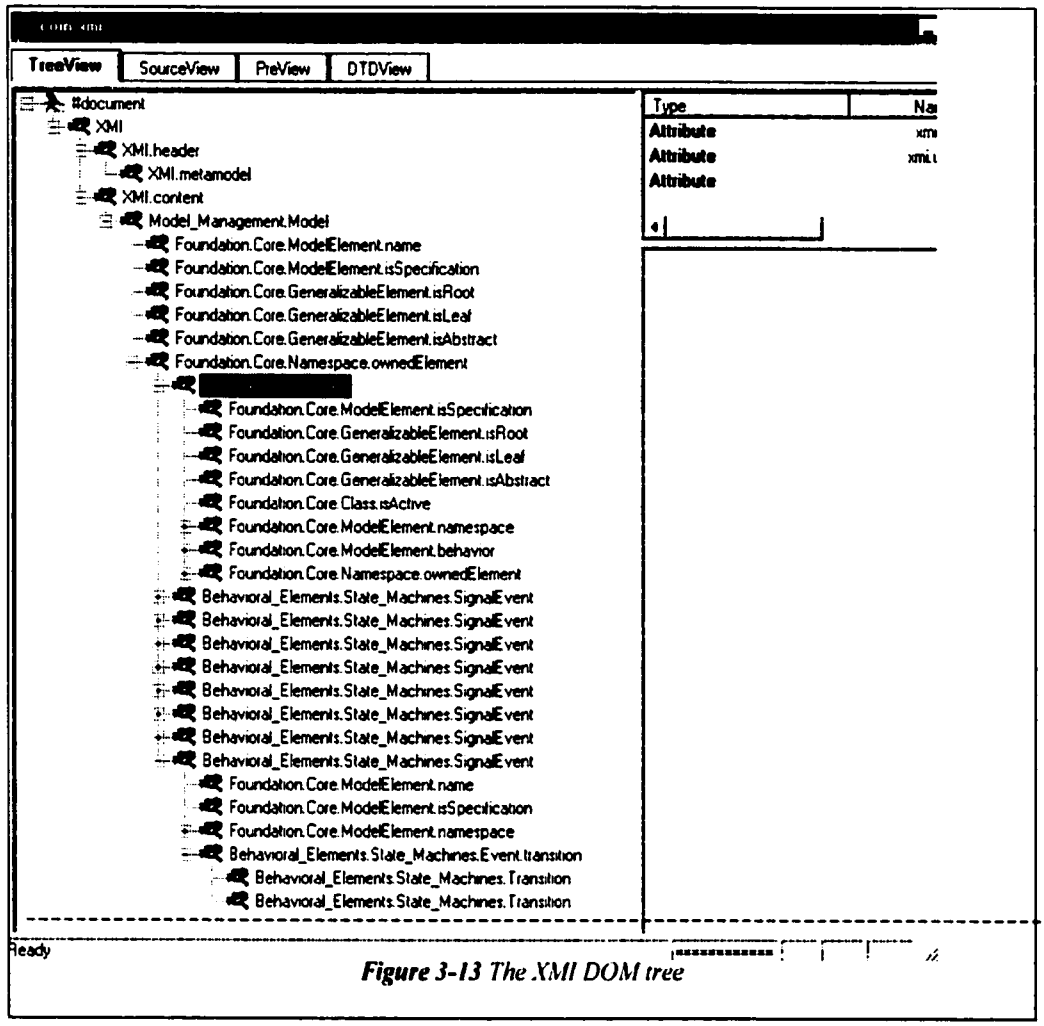


***Figure 3-13** The XMI DOM tree*

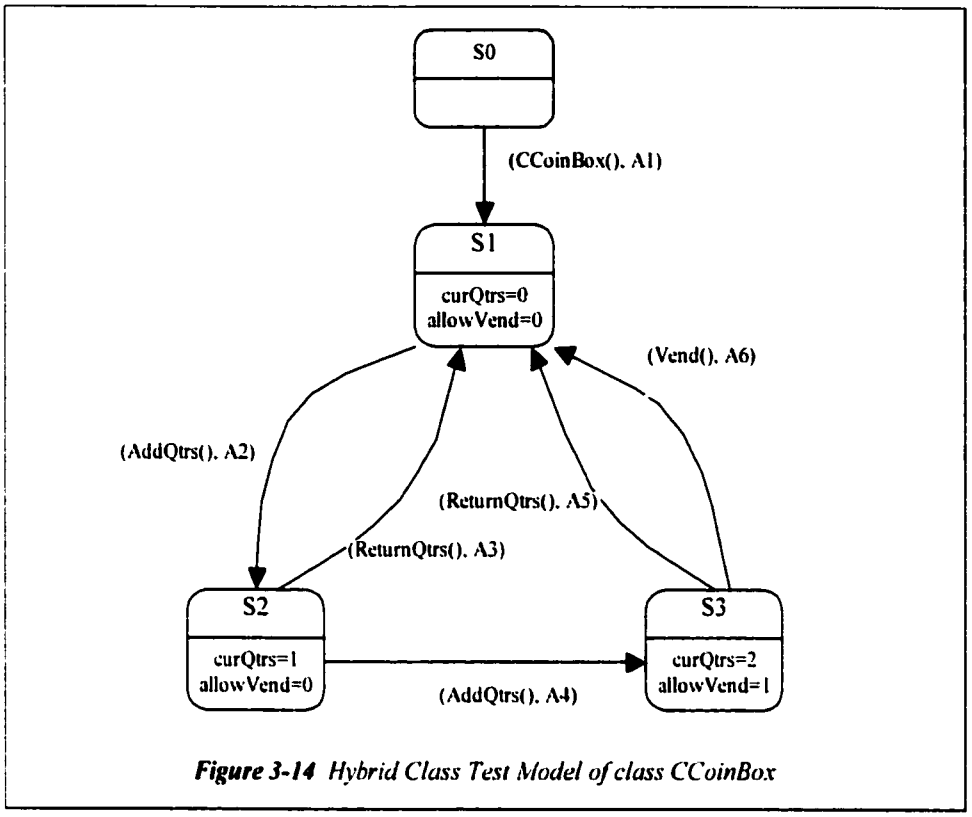


Figure 3-14 *Hybrid Class Test Model of class CCoinBox*

Assume the CCoinBox UML design specification is saved as XMI format (see Figure 3-12). The testing tool parses the XMI format design specification, builds a DOM tree based CUT state diagram (see Figure 3-13). On the class state diagram, the respective def/use sequences are then added to the transitions. The obtained new diagram (see Figure 3-14) is the hybrid class test model of class CCoinbox. Following the description,

where

S = {S0, S1, S2, S3}

V = {curQtrs, allowVend}

F is defined as follows:

F(S0) undefined

F(S1)(curQtrs)=0

F(S1)(allowVend)=0

F(S2)(curQtrs)=1

F(S2)(allowVend)=0

F(S3)(curQtrs)=2

F(S3)(allowVend)=1

M={CCoinBox(), AddQtr(),ReturnQtrs(),Vend()}

→ contains

$$S0 \xrightarrow{CCoinBox()} S1$$

$$S1 \xrightarrow{AddQtr()} S2$$

$$S2 \xrightarrow{AddQtr()} S3$$

$$S2 \xrightarrow{ReturnQtrs()} S1$$

$$S3 \xrightarrow{ReturnQtrs()} S1$$

$$S3 \xrightarrow{Vend()} S1$$

→∈S × L × S contains the transitions. Each transition l∈L is a pair of the name of the method to trigger the transition and a list of definition/use of data members in the the CUT.

According to the state values given in the class state diagram, the below def/use sequences are obtained from the source code.

A1: [(def, curQtrs), (def, allowVend), (def, totalQtrs)]

A2: [(use, curQtrs), (def, curQtrs), (use, curQtrs)]

A3: [(def, curQtrs)]

A4: [(use, curQtrs), (def, curQtrs), (use, curQtrs), (def, allowVend)]

A5: [(def, curQtrs)]

A6: [(use, allowVend), (use, curQtrs), (use, totalQtrs), (def, curQtrs), (def, totalQtrs), (def, allowVend)]

➢ The Transition Tree of Class CcoinBox

Using the algorithm in section 3.6, the transition tree of the *CCoinBox* class can be built. The structure of the tree is depicted in Figure 3-15. Each node represents a state of the *CCoinBox* object. It contains a method, state values and the corresponding data member occurrences in the method.

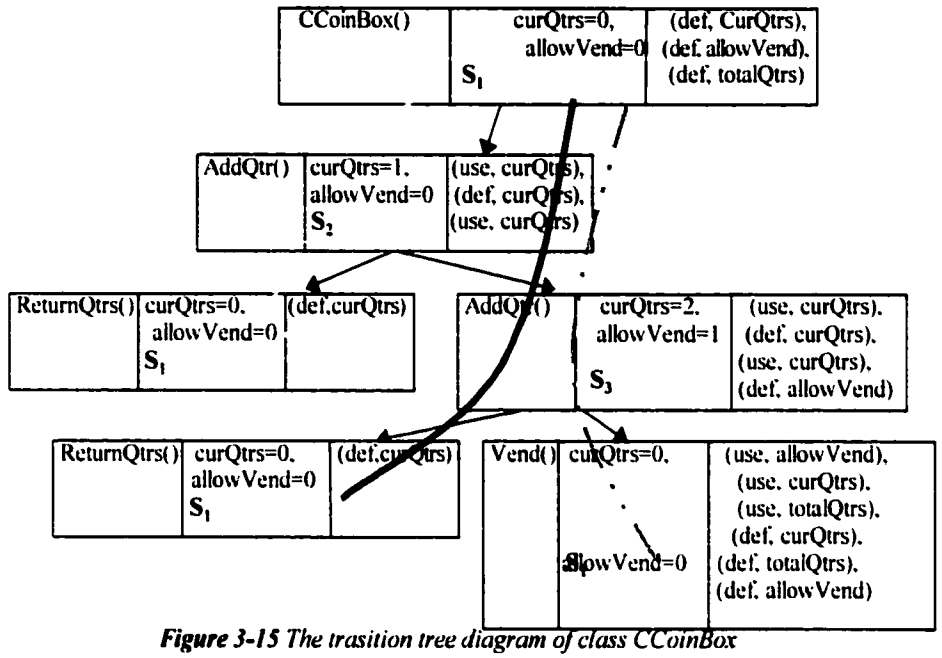


Figure 3-15 *The trasition tree diagram of class CCoinBox*

➢ State-based Testing and Test Results

The *test message generator* visits each node of the transition tree from the root, along the paths. While traversing the tree test message file for a *CCoinBox* object is produced. A section of the test message file for a *CCoinBox* object (called *CCB*) is listed on the left-hand side of Table 3-1. If the *CCB* object is examined by executing the file, the execution results of test messages are stored in a test result file, which is shown on the right hand side of Table 3-1.

The *test result inspector* can detect the error which occurs in the *ReturnQtrs()* function within the method sequence *CCoinBox()* → *AddQtr()* → *AddQtr()* → *ReturnQtrs()*. This error occurs, because after the sequence of methods is executed the resulting state values (*curQtru = 0, allowVend = 1*) cannot match the expected state values (*curQtru = 0, allowVend = 0*).

***Table 3-1** The test result files of a CCoinBox object*

method Names	State Values
:	:
CCoinBox()	curQtrs = 0, allowVend = 0
AddQtr()	curQtrs = 1, allowVend = 0
ReturnQtrs()	curQtrs = 0, allowVend = 0
:	:
CCoinBox()	curQtrs = 0, allowVend = 0
AddQtr()	curQtrs = 1, allowVend = 0
AddQtr()	curQtrs = 2, allowVend = 1
Vend()	curQtrs = 0, allowVend = 0
:	
CCoinBox()	curQtrs = 0, allowVend = 0
AddQtr()	curQtrs = 1, allowVend = 0
AddQtr()	curQtrs = 2, allowVend = 1
ReturnQtrs()	curQtrs = 0, allowVend = 1
:	:

➢ Data Flow Anomaly Detection

The *def-use info generator* can produce lists of data occurrences within the sequences of methods when it traverses the transition tree of the *CCoinBox* class (in Figure 3-15). The lists of data occurrence for the three sequences of methods of the class are shown in Table 3-2.

The definition-use pairs of each data member in the lists are computed in order to find any data anomalies. The grey bold arcs in Table 3-2 show the pairs of definition-use data members. The black dotted arcs show that data anomalies may occur, for example, when data members are defined without being used.

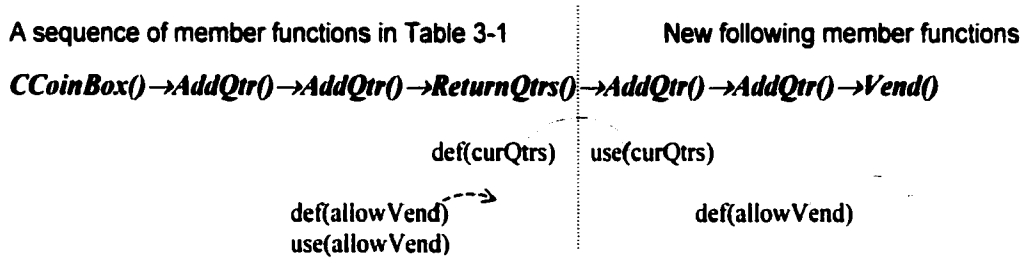
The black dotted arcs in the first row of Table 3-2 show that the *allowVend* and *totalQtrs* data members defined in *CCoinBox()* and the *curQtrs* data member defined in *ReturnQtrs()* are not used within the sequence of methods. However, they may be used within other methods such as *AddQtr()→AddQtr()→Vend()*. The *curQtrs* data member defined in *ReturnQtrs()* can be used in the first *AddQtr()* function, and the *totalQtrs* data member can be used in the *Vend()* function.

Two *allowVend* data members are defined in *CCoinBox()* and the second *AddQtr()* function respectively without any intervening use. This can be found in the second row of Table 3-2. The same data anomaly can be found in the third row of the table. This raises the questions: Is it necessary to define the *allowVend* data member in the constructor? If it is necessary, should it be used before the second definition in the *AddQtr()*? In the third row, an *allowVend* data member (defined in the *AddQtr()* function) and a *curQtrs* data member (defined in the *ReturnQtrs()* function) are not used.

***Table 3-2** The definitions - uses of the data members within sequences of member functions*

	CCoinBox()	AddQtr()	AddQtr()	ReturnQtrs()	Vend()
1	def(curQtrs). allowVend. totalQtrs)	use(curQtrs). def(curQtrs). use(curQtrs)		def(curQtrs)	
2	def(curQtrs. allowVend. totalQtrs)	use(curQtrs). def(curQtrs). use(curQtrs)	use(curQtrs). def(curQtrs). use(curQtrs). def(allowVend)		use(allowVend). def(totalQtrs). use(curQtrs). def(curQtrs). use(totalQtrs). def(allowVend).
3	def(curQtrs. allowVend. totalQtrs)	use(curQtrs). def(curQtrs). use(curQtrs)	use(curQtrs). def(curQtrs). use(curQtrs). def(allowVend)	def(curQtrs)	

As in the discussion of the first row above, the defined *curQtrs* data member can be used in the *AddQtr()→AddQtr()→Vend()* sequence. In the preceding sequence, however, another defined *allowVend* data member follows the unused *allowVend* data member, as computed below.



To remove this data anomaly, should the *allowVend* data member defined in the *AddQtr()* function be used in the following *ReturnQtrs()* function? In other words, should it be used as a predicate use or a computation use?

➢ ***Possible Solution for Removing the Error in Class CcoinBox***

There are several ways to solve the data anomalies of the *allowVend* data member in functions *CCoinBox()* and *AddQtr()*. One method is to reference the *allowVend* data member as a predicate-use based on data flow analysis for achieving all-definition coverage, as shown in the following *ReturnQtrs()* and *AddQtr()* functions. The former replaces the *ReturnQtrs()* function, which only contains a single statement *curQtrs = 0;* see Figure 3-16. Thus the changes are:

```
void ReturnQtrs() {
        if (curQtrs > 0) {curQtrs = 0;}
        if (allowVend > 0) {allowVend = 0;}
}
```

```
void CCoinBox::AddQtr(){
    curQtrs = curQtrs + 1;
    if (curQtrs > 1 && allowVend == 0){
        allowVend = 1; }
}
```

Figure 3-16 The modified ReturnQtrs() and AddQtr() functions for the CCoinBox class

This replacement not only eliminates the data anomaly of the *allowVend* data member but also removes the state-based error discovered previous. If the *ReturnQtrs()* function in Figure 3-16 was coded as above and the sequence of methods *CCoinBox()* → *AddQtr()* → *AddQtr()* → *ReturnQtrs()* were executed, then the resultant state values of a *CCoinBox* object would be *curQtrs == 0* and *allowVend == 0*, rather than the error result *curQtrs == 0* and *allowVend == 1*, shown in Table 3-2. This indicates that the above correction solves the data anomaly and removes the state-based error. Kung et al. in [1996] have eliminated the state-based error by changing the *ReturnQtrs()* function to

void ReturnQtrs() { curQtrs = 0; allowVend = 0;}.

However, this solution does not exclude the data anomaly of the *allowVend* data member. Additionally, it raises another argument: should the *allowVend* and *curQtrs* data members be reset to zero in the function if their values are already zero? For example, if the sequence of functions *CCoinBox()* → *AddQtr()* → *ReturnQtrs()* → *ReturnQtrs()* are applied to a *CCoinBox* object, this can result in switching off a light when it is already off.

Chapter Four

Design and Implementation of Prototype of Test case generation system based on hybrid testing strategy

The design and implementation of Hybrid Test Case Generation System (HTCGS) is based on the hybrid testing strategy and algorithm explained in Chapter 3. It mainly involved the UML design specification transformation and test cases generation module. In HTCGS, testing software gets design specification through parsing XMI file. XMI provides a vendor-independent, platform-independent and language-independent standard based way to exchange design specification between design software and testing software.

4.1 System Requirements

The prototype of Hybrid Test Case Generation System (HTCGS) is implemented using Java, XMI and UML. There are several reasons to choose Java, XMI and UML.

- Java is a general purpose, object-oriented programming language, which provides powerful features for file processing, GUI and string manipulation. Java is platform independent and reusability. Java provides the API with CORBA and XML implementation. In UML 1.3, Java-XMI and UML-XMI mapping are provided. Java is the best choice for implementing hybrid test case generation system.

- UML provides a powerful mechanism for describing software. UML combines the three popular approaches of Booch, Rumbaugh and Jacobson, and has been accepted by the OMG as an industry standard for object-oriented analysis and design notation. It comprises a number of diagrams used to describe different aspects of a system including static, dynamic and use-case views.

- XML Metadata Interchange Format (XMI) specifies an open interchange model, which is intended to give developers working with object and distributed technology the ability to exchange data between tools, applications, repositories, business objects, and programs based on XML. This is a stream-based model interchange format that

enables exchange of modeling, repository, and programming data over the network and Internet in a standardized way. XMI is a much-needed specification to bring consistency and compatibility to applications created in collaborative environment. XMI is a perfect information format for automated test case generation system.

The HTCGS environment for this implementation requires Windows95/NT or UNIX operating system, JDK1.2 or higher version, with Apache group XML parser Xerces, Novosoft's NS UML 0.4.17 or higher.

4.2 Development of XMI parser and State-Transition Table construction module

In our research, we assume that software designer has finished the software design and saved his UML design specification in a XMI format file.

In this thesis, a tool called XMI Parser and State-transition Table construction Module (XPSTM) has been developed for automatically generating State-transition table information from XMI file. It is implemented in four main classes: *state-transition table builder class, XMI parser class, state-transition pair class, and state-transition table class*. XPSTM's class diagram is in figure 4.1. A third part package Novosoft's NS UML is used in this XMI parser implementation, which transfer the basic XMI components to Java class mapping.

Class ***state-transition pair*** stores the information of state, and its incoming transition and output transition relationship.

Class ***State-transition table*** class contains a series of state-transition pair and initial state, current state information.

Class ***State-transition table builder*** manages the XMI parser, and generates a state-transition table for class that have state machines.

XMI parser class heavily handles the work that parses the XMI file, and builds a DOM (Document Object Model) tree of XML file. In my implementation, a third part package NS UML 0.4.13 is used to help parsing the XMI file.

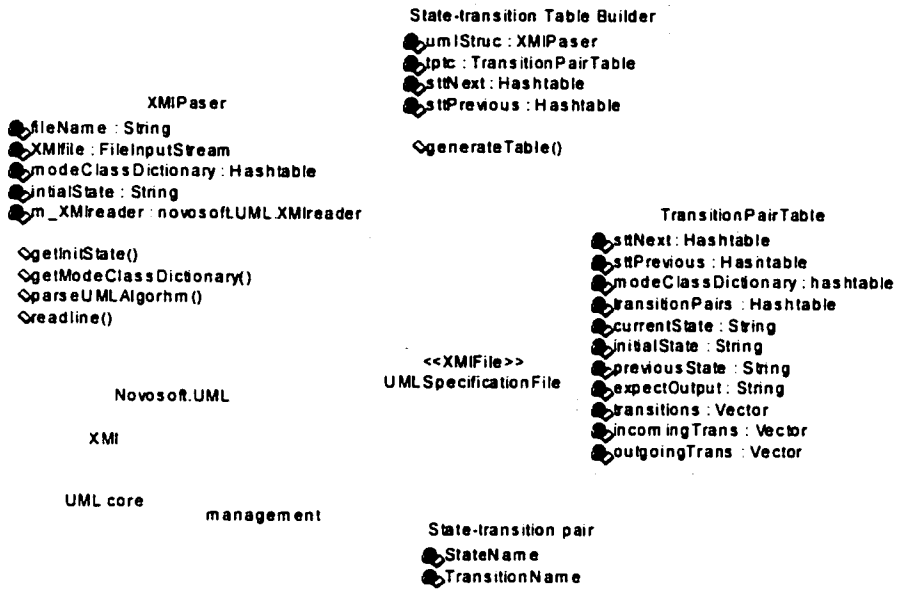


Figure 4-1 the XMI Parser and State-transition Table construction Module

Figure 4-2 gives a XMI DTD (Document Type Definition) fragment about UML State and transitions.

Figure 4-3 describes how to represent the UML state, transition, and state machine in our XMI Parser and State-transition Table construction Module. This is a template-based representation. Figure 4-4 shows a XMI fragment for queue class UML design specification, emphasis the state and transition parts.

Figure 4-5 gives a screen snoop about the generation of State-transition table of Queue class. This table is extracted from the input XMI file.

The State-transition diagram also can be extracted from XMI file. Figure 4-6 shows the queue class State-Transition diagram that is extracted from XMI file by our system.

```
<!-- UML CLASS: State -->
<!ENTITY % StateProperties '%StateVertexProperties;' >
<!ENTITY % deferredEvent 'deferredEvent' >
<!ELEMENT deferredEvent (XMI.reference) >
<!ENTITY % StateAssociations '%StateVertexAssociations;,
                              %deferredEvent;*' >
<!ENTITY % entry 'entry' >
<!ELEMENT entry (#PCDATA | ActionSequence)* >
<!ENTITY % exit 'exit' >
<!ELEMENT exit (#PCDATA | ActionSequence)* >
<!ENTITY % internalTransition 'internalTransition' >
<!ELEMENT internalTransition (Transition*) >
<!ENTITY % StateCompositions '%StateVertexCompositions;,
                              %entry;?,
                              %exit;?,
                              %internalTransition;?' >
<!ELEMENT State (%remoteContent; |
                    (%StateProperties;,
                     %StateAssociations;,
                     %StateCompositions;)) >
<!ATTLIST State %XMI.ElementAttributes;>
<!-- UML CLASS: SignalEvent -->
<!ENTITY % SignalEventProperties '%ModelElementProperties;' >
<!ENTITY % SignalEventAssociations '%ModelElementAssociations;,
                                    %signal;' >
<!ENTITY % SignalEventCompositions '%ModelElementCompositions;' >
<!ELEMENT SignalEvent    <%remoteContent; |
                         (%SignalEventProperties;,
                          %SignalEventAssociations;,
                          %SignalEventCompositions;)) >
<!ATTLIST SignalEvent %XMI.ElementAttributes;>
<!-- UML CLASS: SimpleState -->
<!ENTITY % SimpleStateProperties '%StateProperties;' >
<!ENTITY % SimpleStateAssociations '%StateAssociations;' >
<!ENTITY % SimpleStateCompositions '%StateCompositions;' >
<!ELEMENT SimpleState (%remoteContent; |
                         (%SimpleStateProperties;,
                          %SimpleStateAssociations;,
                          %SimpleStateCompositions;)) >
<!ATTLIST SimpleState  %XMI.ElementAttributes;>
<!-- UML CLASS: StateMachine -->
<!ENTITY % StateMachineProperties '%ModelElementProperties;' >
<!ENTITY % StateMachineAssociations '%ModelElementAssociations;,
                                     %context;?' >
<!ENTITY % top 'top' >
<!ELEMENT top (CompositeState) >
<!ENTITY % transitions 'transitions' >
<!ELEMENT transitions (Transition)* >
<!ENTITY % StateMachineCompositions '%ModelElementCompositions;,
                                     %top;,
                                     %transitions;?' >
<!ELEMENT StateMachine       (%remoteContent; |
                             (%StateMachineProperties;,
                              %StateMachineAssociations;,
                              %StateMachineCompositions;)) >
<!ATTLIST StateMachine  %XMI.ElementAttributes;>
```

***Figure 4-2** The XMI DTD for UML State and transition*

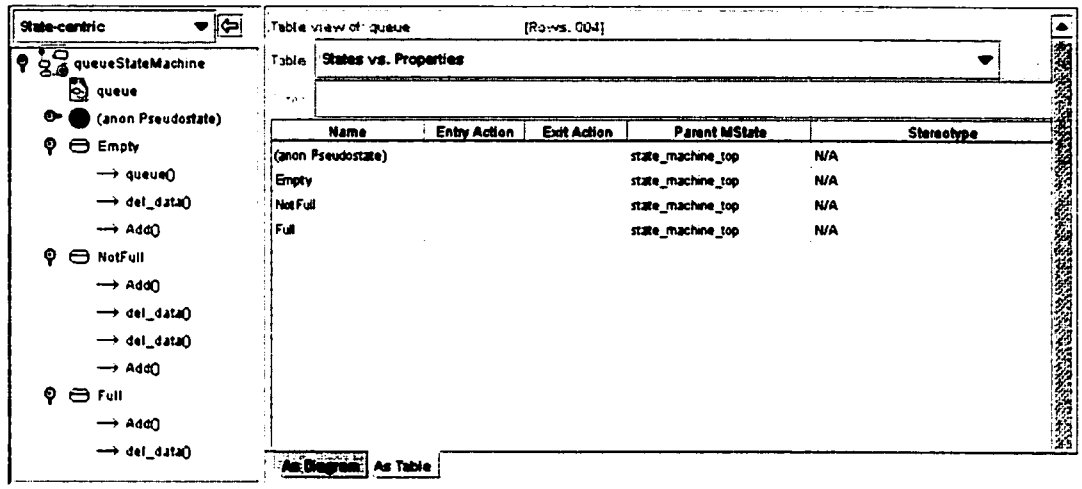
```
<!-- - - - - - - - UML meta-class: State   - - - - - - - -  -->
<macro name="StateProperties">
<![CDATA[
StateVertexProperties
]]>
</macro>
<macro name="StateAssociations">
<![CDATA[
StateVertexAssociations
]]>
</macro>
<macro name="StateCompositions">
<![CDATA[
StateVertexCompositions
<entry>
  <ocl>self.entry</ocl>
</entry>
<exit>
  <ocl>self.exit</ocl>
</exit>
<internalTransition>
  <ocl>self.internalTransition</ocl>
</internalTransition>
]]>
</macro>
<template class="uci.uml.Behavioral_Elements.State_Machines.State">
<![CDATA[
<State XMI.id = "<ocl>self.id</ocl>">
  StateProperties
  StateAssociations
  StateCompositions
</State>
]]>
</template>
<!-- - - - - - - - UML meta-class: SignalEvent   - - - - - - - -  -->
<macro name="SignalEventProperties">
<![CDATA[
ModelElementProperties
]]>
</macro>
<macro name="SignalEventAssociations">
<![CDATA[
ModelElementAssociations
<signal> <XMI.reference /> </signal>
]]>
</macro>
<macro name="SignalEventCompositions">
<![CDATA[
ModelElementCompositions
]]>
</macro>
<template class="uci.uml.Behavioral_Elements.State_Machines.SignalEvent">
<![CDATA[
<SignalEvent XMI.id = "<ocl>self.id</ocl>">
  SignalEventProperties
  SignalEventAssociations
  SignalEventCompositions
</SignalEvent>
]]>
```

***Figure 4-3** The XMI template for UML State-transition*

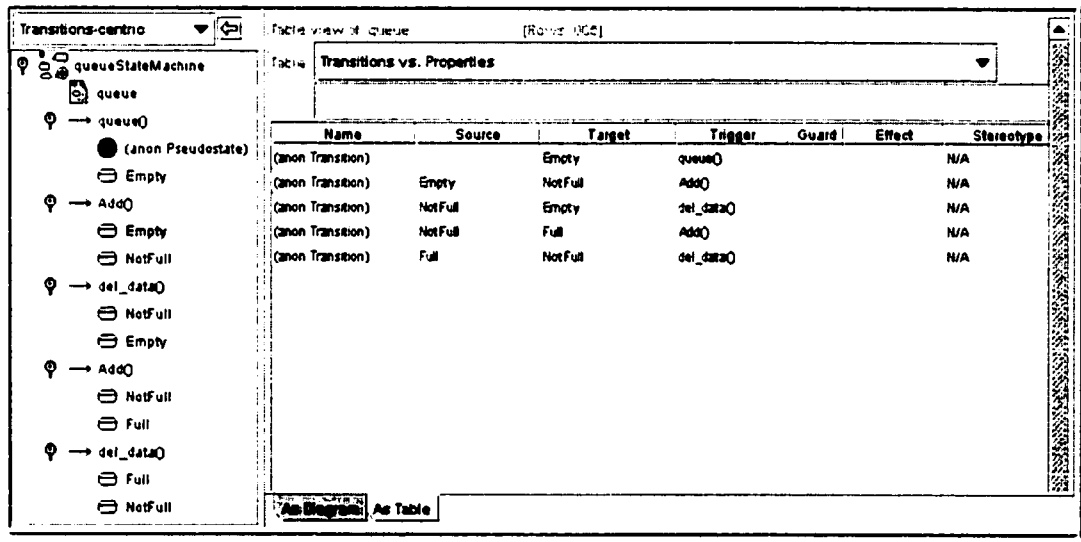
```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi.version="1.0">
  <XMI.header>
    <XMI.documentation>
      <XMI.exporterVersion>0.4.19</XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="1.3"/>
  </XMI.header>
<XMI.content>
    <Model_Management.Model xmi.id="xmi.1" xmi.uuid="127-0-0eb23:e52e988a14:-8000">
      ... <Foundation.Core.ModelElement.name>queue</Foundation.Core.ModelElement.name>
      ...
    <Foundation.Core.ModelElement.namespace>
            <Foundation.Core.Namespace xmi.idref="xmi.1"/>
    </Foundation.Core.ModelElement.namespace>
              <Behavioral_Elements.State_Machines.StateMachine xmi.id="xmi.3"
xmi.uuid="127-0-0-1--6fcbeb23:e52e988a14:-7ffd">
<Foundation.Core.ModelElement.name>queueStateMachine</Foundation.Core.ModelElement.nam
>
...
<Behavioral_Elements.State_Machines.Transition xmi.id="xmi.12" xmi.uuid="127-0-0-1--
6fcbeb23:e52e988a14:-7ff4">
 <Foundation.Core.ModelElement.name>delete</Foundation.Core.ModelElement.name>
  <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
  <Behavioral_Elements.State_Machines.Transition.trigger>
     <Behavioral_Elements.State_Machines.Event xmi.idref="xmi.24"/>
  </Behavioral_Elements.State_Machines.Transition.trigger>
  <Behavioral_Elements.State_Machines.Transition.stateMachine>
     <Behavioral_Elements.State_Machines.StateMachine xmi.idref="xmi.3"/>
  </Behavioral_Elements.State_Machines.Transition.stateMachine>
  <Behavioral_Elements.State_Machines.Transition.source>
     <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.13"/>
  </Behavioral_Elements.State_Machines.Transition.source>
  <Behavioral_Elements.State_Machines.Transition.target>
    <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.10"/>
  </Behavioral_Elements.State_Machines.Transition.target>
  <Behavioral_Elements.State_Machines.Transition.guard>
     <Behavioral_Elements.State_Machines.Guard xmi.id="xmi.25">
     <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
  <Behavioral_Elements.State_Machines.Guard.expression>
     <Foundation.Data_Types.BooleanExpression xmi.id="xmi.26">
     <Foundation.Data_Types.Expression.language>bool</Foundation.Data_Types.Expression.language>

<Foundation.Data_Types.Expression.body>&gt;0</Foundation.Data_Types.Expression.body>
     </Foundation.Data_Types.BooleanExpression>
</Behavioral_Elements.State_Machines.Guard.expression>
<Behavioral_Elements.State_Machines.Guard.transition>
<Behavioral_Elements.State_Machines.Transition xmi.idref="xmi.12"/>
</Foundation.Core.Namespace.ownedElement>
...         </Model_Management.Model>
 </XMI.content>
</XMI>
```

***Figure 4-4** The XMI fragment of class Queue UML design specification*



(a)



(b)

Figure 4-5 State-transition table construct from queue class XMI file
(a) State-view
(b) Transition-view

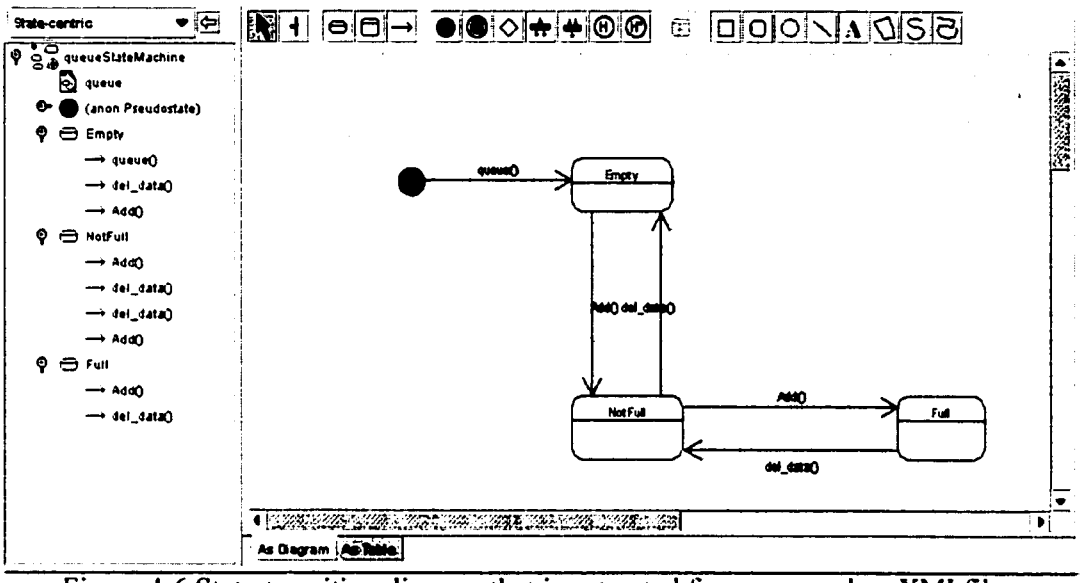


Figure 4-6 State-transition diagram that is extracted from queue class XMI file

4.3 Design and Implementation of Source code Parser and Def-use info Generator

In order to automatically get class method's data occurrence information, a source code parser is developed for static analysis source code, check the source code control flow graph structure.

4.3.1 JavaCC-Generated Source Code Parser

The source code parser and def-use info generator of our environment are developed based on Sun Microsystems Java Compiler Compiler (JavaCC) and Visitor design pattern.

JavaCC is a LL parser generator for Java, compared to the well-known LR parser generator YACC for c. Both JavaCC and the parsers it generates are certified as 100%

pure Java and run on more than forty different platforms without any need for porting the code.

JavaCC generates top-down (recursive descent) parsers as opposed to bottom-up parsers generated by YACC like tools. This allows the use of more general grammars. The lexical specifications such as regular expressions, strings, etc. and the grammar specifications (the BNF) are written together in the same file. It makes grammars easily read and also easily maintain.

The Grammar of the source code parser can be seen in Appendix B.

The package shown in Figure 4-7 contains the class that forms the parser that generates a linked list and tree structure from a Java source files.

The main class JavaParser takes a stream and sends it to the tokenize and then parse it into a tree and returns the root of the tree

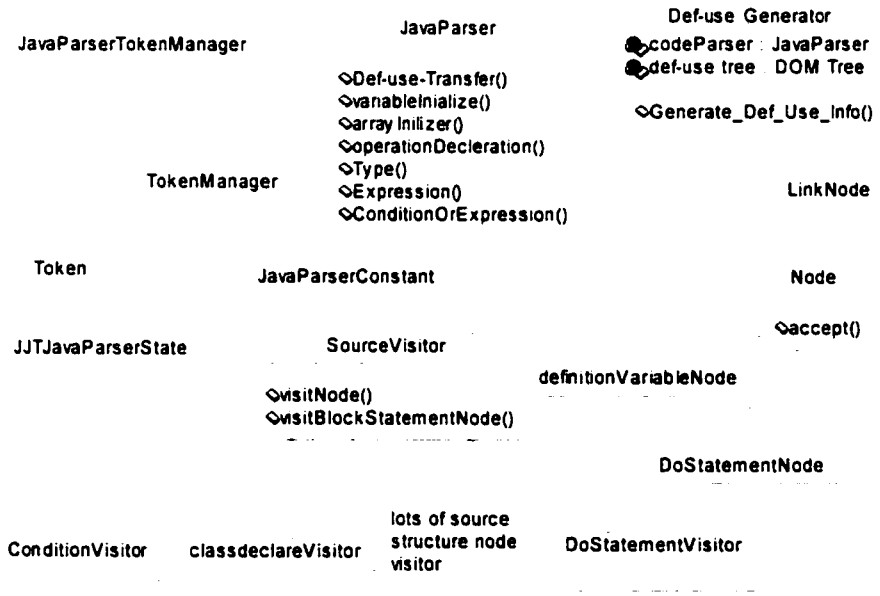


Figure 4-7 JavaCC-generated Source code parser and def-user tree builder

```
PARSER_BEGIN(JavaParser)

public class JavaParser {
 //Parse an input source code stream and produce a tree representing
  public LinkNode parseSource(java.io.DataInputStream in) {
      // Create a parser object
      JavaParser parser = new JavaParser(in);
      // Now parse the input stream
      try {
          rootNode n = parser.transfer();
          // Turn the Java source token-tree into a Def-use Node-tree
          def-userVisitor v1 = new def-userVisitor();
          LinkNode theTree = (LinkNode)n.jjtAccept(v1,null);
          return theTree;
      } catch (Exception e) {
          System.out.println(e.getMessage());
          return null;
      }
   }
 }
PARSER_END(JavaParser)
```

Figure 4-8 *The JavaParser code for source check and build def-use tree*

4.3.2 Symbolic Execution

There are more than one possible Def-Use pairs in each method. as exist condition and switch statements in source code.

We use symbolic execution to capture three relevant aspects of method behavior:

(1) The conditions associated with the execution of paths in the method's control flow.

(2) The relationship between input and output values of a method

(3) The set of variables defined along each path. Both kinds of specifications are expressed as a set of prepositional formulas involving the values of method parameters and component attributes before and after a method is executed.

The information we extract for a class method consists of a set of formulas in the following form:

<precondition> => (<attribute>'=<symbolicexpression>)*

< setofdefinedattributes>

Each precondition is a predicate on attributes and method parameters that leads to the execution of a given path traversing the method. The set of defined attributes includes all the attributes that are defined along such path. The symbolic expression defines the new

value of an attribute after a method is executed. This expression involves method parameters and the old values of the attributes.

Due to the inherent complexity of this kind of static analysis techniques, our algorithms for symbolic execution can take advantage of programmer provided information. Details on these algorithms are referenced in [A. Coen-Porisini etc., 1991], [Ugo Buy etc, 1999].

For our example, the specifications extracted for component CCoinBox are shown in Table4-1,

Table4-1: Path Condition Information extracted for class CCoinBox

CCoinBox()		
(true)	⇒	totalQtr'=0
def={totalQtrs, curQtrs, allowVend}		curQtrs' =0
		allowVend'=0
AddQtrs()		
(curQtrs>1)	⇒	totalQtrs'=totalQtrs
def={curQtrs, allowVend}		curQtrs'=curQtrs+1
		allowVend'=1
(curQtrs==0)	⇒	totalQtrs'=totalQtrs
def={curQtrs}		curQtrs'=1
Vend()		
(allowVend≠0)	⇒	totalQtrs'=totalQtr + curQtrs
def={totalQtrs, curQtrs, allowVend}		curQtrs' = 0
		allowVend'=0
(allowVend==0)	⇒	allowVend'=0
def={ }		
ReturnQtrs()		
(true)	⇒	totalQtrs'=totalQtr
def={curQtrs}		curQtrs'=0

4.3.3 XML Support

With the Extensible Markup Language (XML) becoming the emerging standard for data interchange, we build the XML support for Def-Use info result from JavaParser.

Adapting our Source code Parser into a Def-Use XML parser is a relatively straightforward task. Since only the syntax has changed, but not the semantics of the production rules is still the same.

For Example, def-use info generated from CCoinBox class

```
{ Method CCoinBox( ),
   {[condition, true],
    [(def, curQtrs), (def, allowVend), (def, totalQtrs)]}
}
{ Method  AddQtrs( ),
   {[condition, (curQtrs>1)],
   [(use, curQtrs), (def, curQtrs), (use, curQtrs), (def, allowVend)]}
   {[condition, (curQtrs==0)],
    [(use, curQtrs), (def, curQtrs)]}
}
{ Method ReturnQtrs( ),
   {[condition, true],
    [(def, curQtrs)]}
}
{ Method Vend( ),
  {[condition, (allowVend≠0)],
   [(use, allowVend), (def, totalQtrs), (use, curQtrs), (def, curQtrs), (use, totalQtrs), (def, allowVend)]}
 {[condition, (allowVend==0)],
   [ ]}
}
```

Transfer to Def-Use XML:

```
<CCoinBox>
  <CCoinBox__method>
    <condition  attr="true">
    <def> curQtrs </def>
    <def> allowVend </def>
    <def> totalQtrs </def>
     </condition>
  </CCoinBox__method>
  <AddQtr__method>
   <condition curQtrs&gt"1">
    <use> curQtrs </use>
    <def> curQtrs </def>
    <use> curQtrs  </use>
    <def> allowVend </def>
   </condition>
  <condition curQtrs="0">
    <use> curQtrs </use>
    <def> curQtrs </def>
   </condition>
  </AddQtr__method>
```

```
  <ReturnQtrs_method>
  <condition attr="true">
    <def> curQtrs </def>
   </condition>
  </ReturnQtrs_method>
  <vend_method>
   <condition  allowVend="1">
    <use> allowVend </use>
    <def> totalQtrs </def>
    <use>  curQtrs  </use>
    <def> curQtrs  </def>
    <use> totalQtrs </use>
    <def> allowVend </def>
   </condition>
   <condition allowVend="0">
   </condition>
  </vend_method>
</CCoinBox>
```

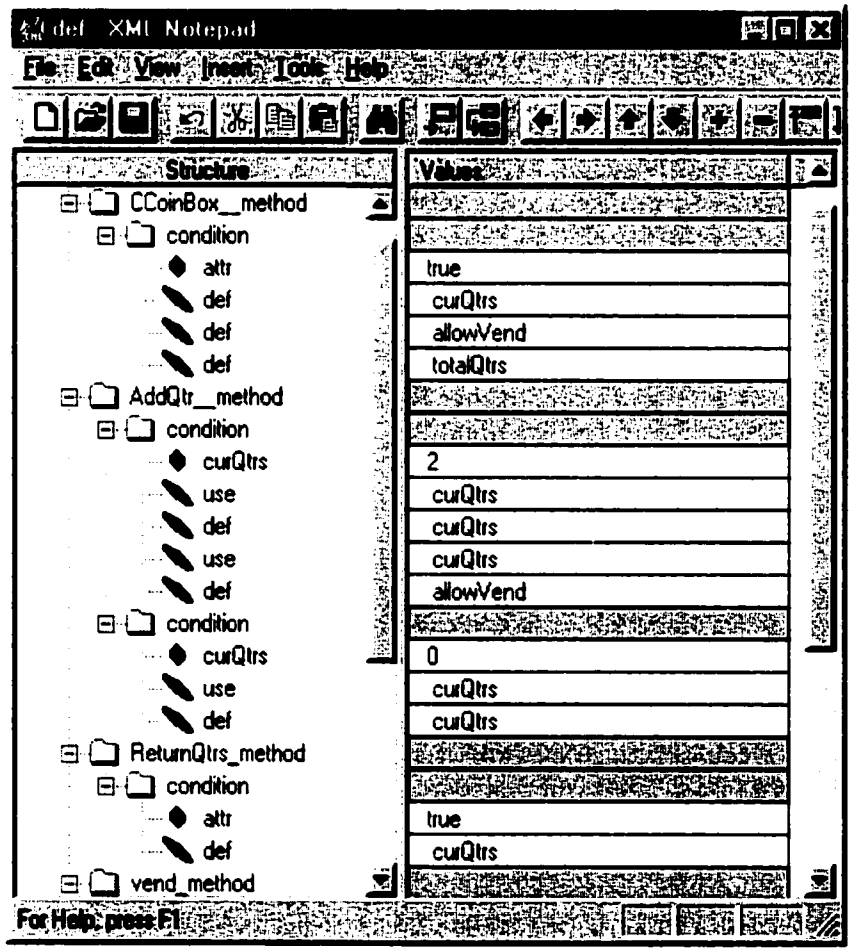


Figure 4-9 The Def-Use info XML file generated from CCoinBox class

4.4 Development Hybrid Class Test Model Navigation and Def-Use info Insertion Algorithm

After State-transition Table and Def-Use info tree have been built, a hybrid class test model navigation and Def-Use info insertion algorithm is implemented.

This algorithm is used to generate the state-based test cases (a set of method sequences), and for each test case, generated a unique def-use pairs for data flow analysis. In this algorithm, we assume that all Def-use info condition variables depend on class state variables.

Figure 4-10 shows the algorithm.

The Hybrid Class Test Model Navigation and Def-use info insertion Algorithm

BEGIN

Step1 The initial state in Hybrid Class Test Model as the root node

Step2 Examine the state that corresponds to each non-terminal leaf node in the tree and each outbound transition on this state. At least one new edge will be drawn for each transition.

Step3 For each edge and node drawn in step2:

a. Note the corresponding transition event on the new branch
b. Find the transition name which equals the def-use info method's name
c. Calculate the condition Boolean value according to the class state value
d. Select a unique sequences of def-use pairs to this transition on this branch
e. if the state that the new node represents is already represented by another node or is a final state, mark this node as terminal, no more transition are drawn from this node

Step4 Repeat step2,3 until all leaf nodes are marked terminal

END

***Figure 4-10** The Hybrid Class Test Model Navigation and Def-use info insertion Algorithm*

For example, the hybrid class test model navigation algorithm generate a test case:

Coinbox() {} → AddQtrs() {curQtrs=0, allowVend=0}→ AddQtrs(){curQtrs=1, allowVend=0} →ReturnQtrs(){curQtrs=0, allowVend=1}

Then response to this test case, can get a unique sequence of def-use pairs like:

[(def. curQtrs), (def, allowVend), (def, totalQtrs)], (because, condition always true)

[(use, curQtrs), (def, curQtrs)], (because , condition curQtrs==0 true)

[(use, curQtrs), (def, curQtrs), (use, curQtrs), (def, allowVend)], (because, condition curQtrs==1 true)

[(def, curQtrs)], (because, condition always true)

4.5 Prototype of Test Cases Generation and Its GUI

Section 4.2 introduces how to transfer a XMI file into a State-Transition Table. Section 4.3 describes from source code generation global Def-Use info XML file. Section 4.4 discusses how to insert the Def-Use info to Class State-Transition Table to get Hybrid Class Test Model, and navigate this model to get test cases.

After we have integrated previous implementation, a Test Cases Generation prototype is built.

Figure 4-11 shows the overview of prototype system, which gives a view of the source code of CUT. Figure 4-12 is a screen capture for the prototype system that has generated the test cases for CCoinbox example.

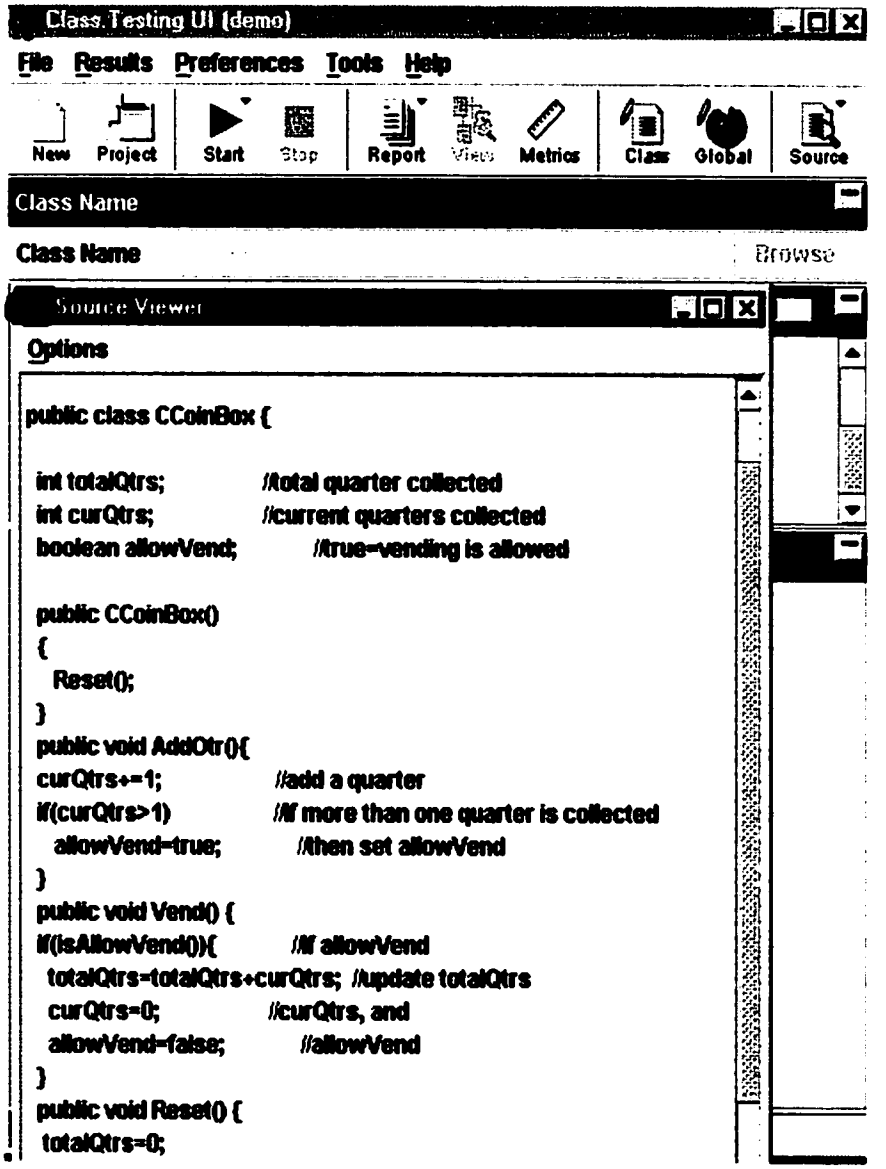


Figure 4-11 Test Cases Generation Prototype GUI for view Source Code

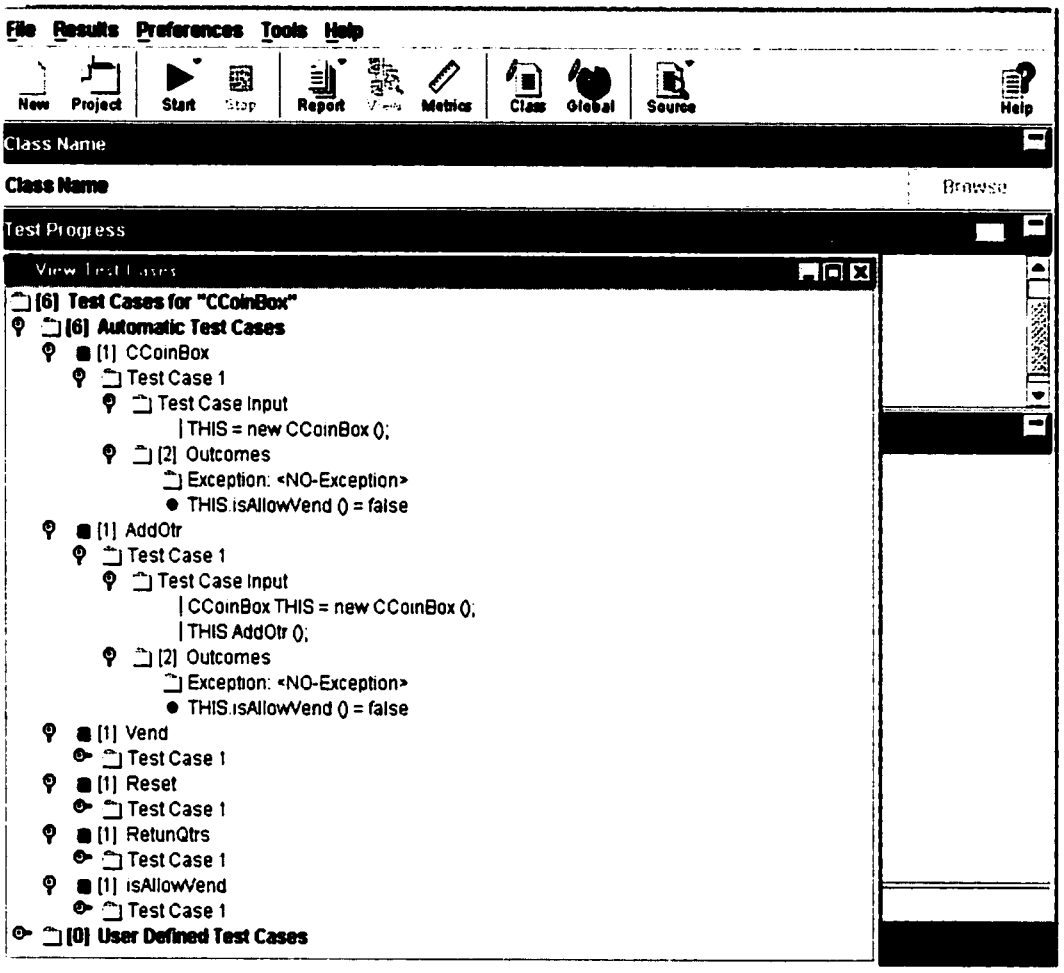


Figure 4-12 Test Cases Generated by Prototype system

Chapter Five

Methodology Evaluation

Harrold and Rothermel [1994] employ their data flow testing technique on a **symbol table class** (called *SymbolTable*) and show how to generate test cases for inter-method and intra-class testing.

5.1 A Symbol Table Class

Since specification-based testing approaches may not provide sufficient code coverage, Harrold and Rothermel [1994] suggest a technique for class testing that supports conventional data flow testing for the data flow interactions in a class. The technique is listed as follows:

(1) A graph representation, called a class control flow graph, is developed for the class under test. In the class control flow graph, the connections of all methods in the class are depicted.

(2) The data flow information is computed from the graph for data flow testing.

In order to explain their approach for class testing, Harrold and Rothermel [1994] propose the *SymbolTable* class, coded in C++, as an example. The class contains the *SymbolTable*, *~SymbolTable*, *AddtoTable* and *GetfromTable* public methods and six private methods: *Lookup*, *Hash*, *GetSymbol*, *GetInfo*, *AddSymbol*, and *AddInfo*.

The calls from public methods to private functions and interactions among private functions in the *SymbolTable* class, show that it is a suitable example for demonstrating their testing technique in intra-method and inter-method testing. Except for the constructor and destructor, there are only two public methods in the class. It is a simple example to show the determination of sequences of public method calls in an arbitrary order, based on the data flow criteria. It is a good example to show that the hybrid class testing technique can also be employed for testing the classes, which have complicated connections of public and private functions.

Case Description

The implemented *SymbolTable* class is shown in Figure 5-1. A symbol table is usually used throughout a compilation to build up information about the names of identifiers used throughout a source program. While compiling a program, the compiler must record the names (including type, scope, and memory assignment) of all declared identifiers. When the compiler parses an identifier outside of a declaration statement, it checks to see whether the identifier has been declared. If it has, the compiler looks up the appropriate information in the symbol table. If it has not, the compiler gives an "undefined symbol" error message [Weiss, 1998]. A symbol table usually has enough storage (implemented via a linked-list, a binary tree or an extensible table) for storing all the possible identifiers in a source program.

```
//symboltable.h: definition of Symboltable class
#include "symbol.h"

class SymbolTable {
private.
      TableEnter *table;
      Int numentries, tablemax;
      Int *Lookup (char *);
Public;
      SymbolTable (int n){
          tablemax = n;
          numentries = 0;
          table = new TableEntry[tablemax]; };
      ~SymbolTable() { delete table;};
      int AddtoTable (char *symbol, char *syminfo);
      int GetfromTable (char *symbol, char *syminfo);
};
// symboltable.c: implementation of SymbolTable class
#include "symboltable.h"

int SymbolTable :: Lookup(char *key, int index){
      int saveindex;
      int Hash(char *);
      saveindex = index = Hash(key);
      while (strcmp(GetSymbol(index), key) !=0) {
          index++;
          if (index == tablemax) /*wrap around */
              index = 0;
          if (GetSymbol(index)==0 || index==saveindex)
              return NOTFOUND;
      }
      return FOUND;
}
```

```
int SymbolTable :: AddtoTable (char *symbol, char
*syminfo) {
      int index;
      if (numentries < tablemax) {
          if (Lookup(symbol, index) == FOUND)
              return NOTOK;
          AddSymbol (symbol, index);
          AddInfo (syminfo, index);
          numentries++;
          return OK;
      }
      return NOTOK;
}

int SymbolTable :: GetfromTable (char *symbol, char
**syminfo) {
      int index;
      if (Lookup(symbol, index) == NOTFOUND)
          return NOTOK;
      *syminfo = GetInfo(index);
      return OK;
}
void symbolTable :: AddInfo (syminfo, index){
          :
      strcpy(table[index].syminfo, syminfo);
}
char *SymbolTable :: GetInfo(index) {
          :
      return table[index].syminfo;
}
```

***Figure 5-1** Partial listing for the SymbolTable class [adopted from Harrold and Rothermel, 1994]*

Applying Hybrid Testing to Class Symbol Table

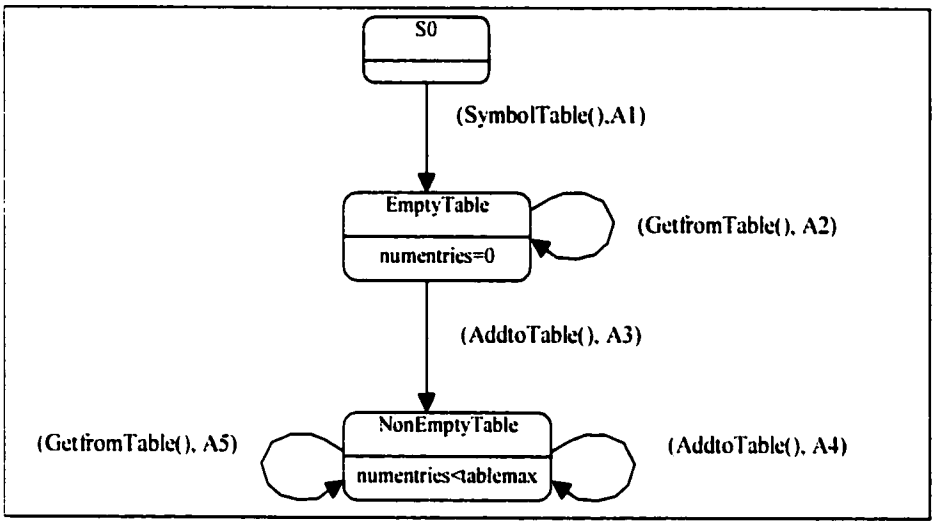


Figure 5-2 *The Hybrid Class Test Model of class SymbolTable*

- **S** = {S0, EmptyTable, NonEmptyTable}
- **V** = {numentries, tablemax, table}
- **F** is defined as follows:

  F(S0) undefined

  F(EmptyTable) (numentries)=0

  F(NonEmptyTable)(numentries)< tablemax

- **M** ={ SymbolTable(), AddtoTable(), GetfromTable()}
- → contains

$$S0 \xrightarrow{SymbolTable()} EmptyTable$$

$$EmptyTable \xrightarrow{GetfromTable()} EmptyTable$$

$$EmptyTable \xrightarrow{AddtoTable()} NonEmptyTable$$

$$NonEmptyTable \xrightarrow{AddtoTable()} NonEmptyTable$$

$$NonEmptyTable \xrightarrow{GetfromTable()} NonEmptyTable$$

According to the state values given in the class state diagram, the below def/use sequences are obtained from the source code of CUT.

A1: [(def, numentries), (def, tablemax), (def, table)]

A2: [(use, table), (use, tablemax), (use, table)]

A3: [(use, numentries), (use, tablemax), (use, table), (use, tablemax), (def, table), (use, numentries), (def, numentries)]

A4: [(use, numentries), (use, tablemax), (use, table), (use, tablemax), (def, table), (use, numentries), (def, numentries)]

A5: [(use, table), (use, tablemax), (use, table)]

The Transition Tree of Class Symbol Table

The *Lookup()*, *GetSymbol()*, *AddSymbol()*, and *Addinfo()* functions are directly or indirectly called by the *AddtoTable()* function. The data members, *table*, *numentries* and *tablemax* are defined and used within those functions. A similar situation occurs when the *GetfromTable()* function is executing.

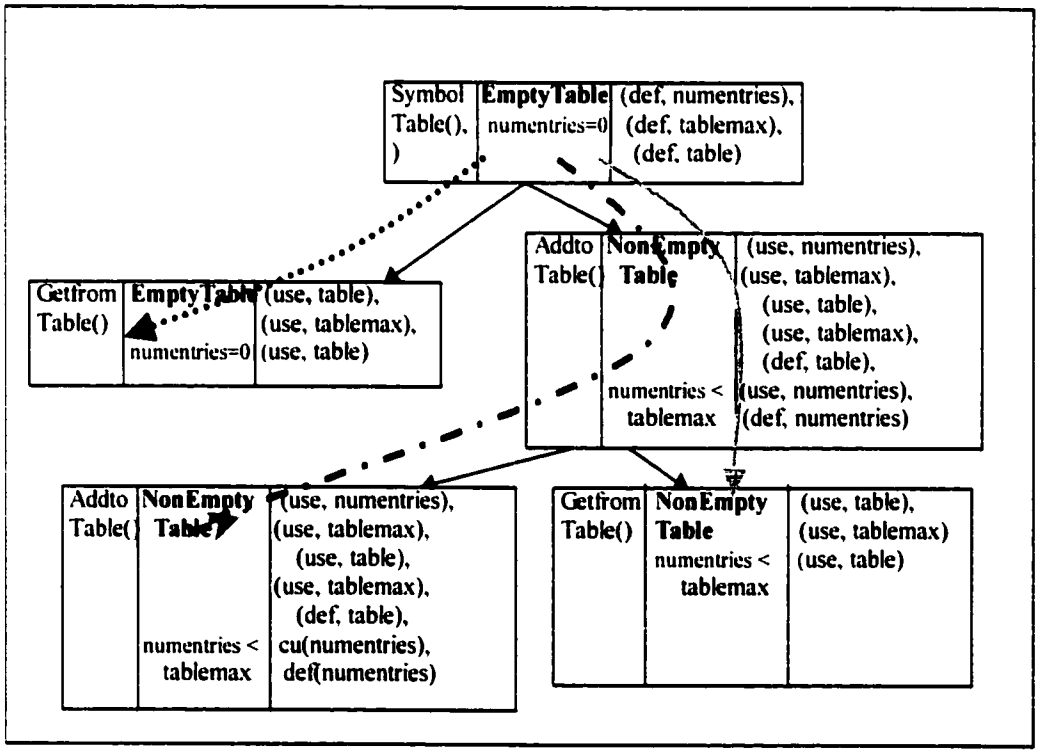


***Figure 5-3** The transition tree of class SymbolTable from hybrid testing*

Test Messages

Examples of sequences of methods in the file are listed as follows. The implemented *SymbolTable* class passes the testing by executing the sequences of test messages.

SymbolTable()→GetfromTable();

SymbolTable()→AddtoTable()→AddtoTable()→GetfromTable();

SymbolTable()→GetfromTable()→AddtoTable()→AddtoTable();

SymbolTable()→AddtoTable()→GetfromTable()→GetfromTable()→AddtoTable()

The Def-Use Information of the Data Members

Before computing the definition-use pairs within the sequences of test messages, the speciality of each data member of the class is discussed as follows:

- The *tablemax* data member is defined with an initial value in the constructor. It is used in *AddtoTable()* and *Lookup()* as a predicate use.
- Data member *numentries* is initialized (defined) with a zero value in the constructor, and its value is increased after the *AddtoTable()* function is executed. It is used and defined in the *AddtoTable()* function.
- An array format data member *table* contains *n* (where *tablemax* = *n)* different table entries. When computing definition-use pairs, each individual entry is treated as a single variable. Therefore, each entry is:

(a) defined with an initial value when the table is created,

(b) checked whether it is empty before an identifier is added into it, so that each table entry is used in the *GetSymbol()* function before it is defined in the functions *AddSymbol()* and *AddInfo()*.

Based on the above discussion and using the data flow technique of the hybrid testing method, no data anomaly was found within the *SymbolTable* class at intra-class testing level.

Discussion

The *SymbolTable* class has only two public methods, and there are only four discretionary sequence methods. It is not so difficult to pick up feasible sequences from the sequence functions. The intra-class sequences of methods <*AddtoTable*,

AddtoTable> and <*AddtoTable*, *GetfromTable*> are selected by computing intra-class def-use pairs in [Harrold and Rothermel, 1994]. The generation of sequences of intra-class test messages using the state-based technique of hybrid testing is easier than using data flow testing.

When selecting sequences of methods for the *SymbolTable* class, Harrold and Rothermel state: "*.... This suggests that* ***GetfromTable*** *cannot affect* ***AddtoTable****, and that we do not need to test method sequence* <***GetfromTable, AddtoTable***>".

Nevertheless, in practice a *GetfromTable* message that is followed by an *AddtoTable* message may be sent to a *SymbolTable* object. For example, the variable *i* was not declared before *if (i < x)* condition statement. The compiler responds with the error message "*Error TestSyntax.cpp 6: Undefined symbol 'i'*." This example shows that it is possible to send a *GetfromTable* message to a symbol table object before sending an *AddtoTable* message to it.

This case study indicates that code-based testing may not generate enough test cases and the requirements of specifications should also be considered. The hybrid-testing tool serves these two approaches, to improve traditional class testing.

Evaluation

No known state-based errors or data anomalies were found in the *SymbolTable* class. Hybrid testing can also produce all definition-use information within both the public and private functions of a class.

Harrold and Rothermel in [1994] adopt existing data flow testing techniques to generate test cases for intra-method, inter-method and intra-class testing upon the *class control flow graph*. This graph depicts the connection of all the methods of the class under test. Data anomaly detection is not discussed in [Harrold and Rothermel, 1994]. Table 5-1 shows the evaluation of the hybrid class testing method using the *SymbolTable* class. No known errors were found by the hybrid method, but it can generate more test cases for intra-class testing than the method of Harrold and Rothermel.

***Table 5-1** The evaluation of the hybrid class testing method using the SymbolTable class*

Testing Method	Class *SymbolTable*		Technique	Error Removal Suggestion
	State-based Errors	Data Anomalies		
The Hybrid Method	Not found	Not found	• More test cases generated for intra-class testing by using the state-based technique • Data anomaly detection	No
H & R's Method	No execution	No execution	Test case generation based on data flow criteria	No

5.2 Summary of the four class testing techniques

For evaluation the hybrid testing strategy, three existing cases from previous research work are adopted. The benefit of adopting and adapting existing cases from research work is that the cases provide an opportunity to carry out comparisons. Another major benefit of using existing case studies is that there is an understanding of the error domain under study.

A **coin box class**, called *CCoinBox*, presented by Kung et al. in [Kung, 1996] demonstrates their object state testing technique for finding state-based errors. In using this class I would like to seek to establish that hybrid-testing model is capable of finding the same state-based errors as the technique of Kung et al., as well as other errors such as data anomalies. This example can be seen in case study of Chapter 3.

Harrold and Rothermel in [Harrold and Rothermel, 1994] employ their data flow testing technique on a **symbol table class** (called *SymbolTable*) and show how to generate test cases for inter-method and intra-class testing. This class is selected to demonstrate whether hybrid-testing model can generate the same (or more) test cases, in order to show whether hybrid class testing can be considered as being better than data flow testing in intra-class test case generation. The private member functions of the *SymbolTable* class contain many intra-class definition-use pairs. This class therefore provides an

opportunity to determine whether hybrid-testing model has the ability to detect data flow anomalies.

Hong et al. [Hong, 1995] propose the *class flow graph* (CFG) of the ***Door* class** to demonstrate their data flow testing technique for generating all definition coverage test cases. In adopting this class I am seeking to reproduce these results using hybrid-testing model. In addition, the data flow technique of the hybrid method can further be used to detect whether any data anomalies exist in the generated test cases.

***Table 5-2** Summary of the four class testing techniques*

	The hybrid testing technique	**Kung et al.'s technique[30]**	**Harrold and Rothermel's technique[18]**	**Hong et al.'s technique[20]**
Test model	Hybrid Testing Model combined State-based testing and data flow analysis.	Object State Diagrams, constructed by symbolically executing the member functions of the classes.	Class Control Flow Graphs, connecting all methods in classes	Class Flow Graphs, flow graphs represent both the control and data flow of classes
Test Units	Class	Class	Class	Class
State view	Combination of attribute values	Combination of attribute values		Combination of attribute values
Approach	Specification-based and code-based	Code-based	Code-based	Specification-based
Testing Technique	State-based fault detection and data anomaly analysis	State-based fault analysis	Data flow testing	Data flow testing
Test cases generation	Transition tree, derived from hybrid test model	From test model	Computing def-use pairs	From test model
Test Case Format	Sequence of methods and def-use pairs	Sequence of methods	Sequence of methods	Sequence of methods
Measure of Quality	All states and transitions coverage, all definition coverage at intra-class level	All states and transitions coverage.	All-use coverage	All definition coverage

Chapter Six

Related Work

This chapter gives a brief review of earlier work on test cases generation based on state-based testing and data flow testing, including related work on generation test case from UML specification, test case generation from source code, and data anomaly testing technique. Then we describe the advantages of our hybrid testing approach.

6.1 Test cases generation from UML specification

In this section, we review two types of test case generation research based on UML specification.

6.1.1 UMLTest (Rose-based Test Data Generation Tool)

Jeff Offutt and Aynur Abdurazik [1999, 2000] presented an approach to generate test cases from UML State diagram. They built a Rational Rose-based Test Data generation tool (UML Test), parsed Rational Rose specification file (called an MDL file) to get the semantic meanings of the specifications. Then according to different coverage criteria, generate state-based testing cases.

In their work, as the first step, they parse Rose MDL file to get semantic meaning. Rational company doesn't provide open documentation about MDL format. In their paper, there are some assumptions about MDL format. Further there are a lot of CASE tools that can build software design specification, but with different serialization format. MDL is a vendor dependent format.
Compared with our solution, we use XMI format to transfer UML information. As XMI is vendor independent and system independent, our solution can be easily adopted to different UML-based CASE tools.

6.1.2 EFSM-based Test case generation from UML State diagram

Young Gon Kim et al [1999] presented a specification-based approach to class testing using UML state diagrams. A set of coverage criteria is proposed based on control and data flow in UML state diagrams. Test cases are generated satisfying these criteria from UML state diagrams. There are two steps in their solution. First, control flow is identified by transforming UML state diagrams into extended finite state machines (EFSMs). The hierarchical and concurrent structure of states is flattened and the broadcast communication is eliminated in the resulting EFSMs. Second, data flow is identified by transforming EFSMs into flow graphs. Conventional data flow analysis techniques can be applied in flow graphs. The approach transforms UML state diagrams to flow graphs, and then takes conventional flow analysis to generate test cases from flow graphs. The bridge between UML state diagram and flow graphs is extended finite state machines (EFSMs). In the paper, the authors did not mention an automated environment to support the total process of test cases generation. In their approach, they apply existing data flow techniques and coverage criteria. Test cases can be generated as a set of paths that cover the associations between the definitions and uses of each variable and state in UML state diagrams.

6.2 Test Cases Generation using State-based Testing Strategy

In this section, three researches about State-based testing strategy are reviewed.

6.2.1 Threaded multi-way tree

Tsai et al [1999] introduced a State-based testing strategy, using a threaded multi-way tree concept. They followed Chow [1978] work, testing techniques using Finite State Machines (FSMs). A threaded multi-way tree is used to represent the Loop State. The testing solution proposed by Tsai, need manually input the testing specification.

6.2.2 Object State Diagram (OSD)

Kung et al. [1993 and 1996] introduced an object state test model based on an *object state diagram* (OSD). They proposed a reverse engineering approach to automatically derive an OSD from the C++ source code. They extracted states and transitions from the result of the symbolic execution of each method in the class. In an OSD, a state of an object is defined by the value range of a subset of the data members in the object, and state transitions are defined by the execution of a method. A test tree, constructed from the OSD, is then used to generate test cases for testing the state behavior. The disadvantage of this state-based testing method is the cost of creating object models (e.g. state machines) from the code. Assume, for example, a class has N state defining data members and each data member has three states, such that the value of a data member has three intervals (e.g. $k < 0$; $0 \leq k \leq x$; $k > x$). The total number of possible combined states are 3^N. For inspecting test results, they adopted the *fault tree analysis* technique. This is quite different from the state-based testing, which is used for generating test cases in their papers. The fault tree analysis which *"requires expert knowledge which is difficult to find and usually subjective"* [Kung et al., 1996].

6.2.3 State Model for specification-based testing

McGregor and Korson [1994] proposed a specification-based testing approach based upon design states. A state model of a class built directly either by examining the class under test or from an analysis/design specification is required. Additionally, they defined design states on some set of attribute values. Methods were modeled as events. A transition in the model is defined by input parameters and the current object state, a method, and method output parameter and the target object state. They use *design* states to describe the changes in behavior of the object class. The main idea of their technique is: *"For functional testing, the goal is to prove that the software performs in conformance with its specification. In order to achieve that goal, test cases are constructed, executed and the results compared with expected behavior"*. In their

technique, however, each test case (a sequence of messages to an object) and the test results still rely on a human oracle and the state model.

6.3 Test Case Selection Using Data Flow Testing Strategy

Harrold and Rothermel [1994] proposed an approach for class testing based on data flow testing. This technique uses interprocedural data flow analysis on all the methods of a class. They developed a *class control flow graph* to connect all methods in the class, and adapted the *data flow analysis algorithm* to compute the data flow information required for data flow testing. This technique can test the interaction of an object's methods. Since a variable can be defined in one method and used in another, this technique provides a way of selecting sequences of method invocations. For applying data flow testing into classes, they derived a sequence of intra-class test cases from the intra-class def-use information.

Harrold and Rothermel in [1994] define three testing levels (intra-method, inter-method and intra-class) for class testing, and use the *SymbolTable* class as an example to demonstrate their data flow technique. Harrold and Rothermel [1994] also state that the further advantage of their technique is to determine which sequence of methods should be executed to test a class, and point out error sequences with examples that need not be run. However, their class example only has two so that the selected intra-class test cases (based on intra-class def-use pair) are merely pairs public methods of ordered functions. If a class has N public methods, then there are $N!$ sequences at the intra-class testing level. In fact, it is very difficult to select all possible test cases and exclude all infeasible sequences from the $N!$ sequences by referencing the functionality of the class.

6.4 Data Anomaly Testing techniques

Dynamic data flow analysis techniques are addressed in [Chan and Chen, 1987; Huang, 1979]. Chan and Chen propose an automated instrumentation system to perform data flow analysis for Pascal programs. Huang uses a program instrumentation technique for detecting data flow anomalies. In the program instrumentation process, additional statements are inserted into a program for the purpose of information gathering.

Chan and Chen in [1987] describe their dynamic data flow anomaly detection system (AIDA) for Pascal programs. For dynamically carrying out data flow analysis, they adopted the technique of *program instrumentation* [Huang, 1979], which involves the insertion of software probes into the source code of the programs to collect information. In order to initialize, trace or check the states of variables, AIDA transforms the source code to form *instrumented programs*. A *state transition diagram* is used in AIDA to present the state change of each variable and the same diagram is also used in [Huang, 1979]. When a variable enters the abnormal state, this implies a data flow anomaly on the variable [Chan and Chen, 1987].

Static data flow anomaly detection techniques on either intra-procedure or inter-procedure level are discussed in [Fosdick and Osterweil, 1976]. They use path expressions in which the actions in each variable along the program execution paths are expressed, and then the expressions are evaluated relative to each variable [Fosdick and Osterweil, 1976] to efficiently detecting anomalous data flow patterns in a program, which can be represented by a graph first. Then, using static data flow analysis, we can scan the paths entering and leaving each node of the graph to show anomalous data action combinations. The paths of method sequences of a class at the intra-class level can be depicted in the state transition diagram of the class. Thus that static analysis technique is adopted for detecting data anomalies on data members within the sequences of methods.

A practical technique for data anomaly detection is:
(1) Determine whether a data anomaly is present,
(2) Find a path containing the anomaly, and then
(3) Attempt to determine whether the path is executable.
Although infeasible sequence messages can be selected as test cases, performing data anomaly detection on infeasible sequence messages is not necessary. A data anomaly on such a non-executable path is of no concern in this research. The executable sequences of methods (paths) at the intra-class level can be easily generated using state-based criteria.

6.5 Advantages of Our Approach

A hybrid class test model is proposed in our hybrid testing strategy, which contains both the information from specification about the state change of object instances of the Class Under Test (CUT) and the information from the source code about the definition and use of the data members in the CUT. With such an uniformed architecture, we can obtain automated tools to generate test cases for state-based testing and perform data flow testing as the same time. The combination of the two techniques is essential in improving our testing environment and thus, contributes to the enhancement of the reliability of software products.

Hybrid testing strategy keeps the advantage of state-based testing and data flow testing. Hybrid testing generates the data member definition-use information based on class state transition relationship, not just navigating all public method of the whole sequences (Harrold and Rothermel's proposal 1994), and significantly reduces the testing cost.

Hybrid testing can detect some data anomaly that State-based testing cannot trace. In chapter 3, case study gives a good example.

A Standard-based UML information exchange format--- XMI (XML based Metadata Interchange) is used in my research work, to bridge the software designer and software tester. No matter what kind of CASE tools designer use, the testing software can easier understand design specification from different design tools. So hybrid-testing strategy provides a fully automated testing framework for class testing.

Chapter Seven

Conclusions and Future Work

This chapter draws the conclusions and describes possible directions for future work related to this research.

7.1 Conclusions

In this thesis, Hybrid testing strategy is proposed for class testing.

Main contributions of this thesis are:

1. Propose a hybrid class test model, which contains both the information from specification about the state change of object instances of the Class Under Test (CUT) and the information from the source code about the definition and use of the data members in the CUT. With such a uniformed architecture, we can obtain automated tools to generate test cases for state-based testing and perform data flow testing as the same time. The combination of the two techniques is essential in improving our testing environment and thus, contributes to the enhancement of the reliability of software products.
2. The hybrid testing strategy can be used in software design stage and software implementation stage. In design stage, testers can start design test cases. After the coding phase, data flow testing can be applied based on the state information. It need not reengineer to get state information from source code. Expenses of test case generation are reduced significantly.
3. A Standard-based UML information exchange format--- XMI (XML based Metadata Interchange) is used in my research work, to bridge the software designer and software tester. No matter what kind of CASE tools designer use, as long as save as XMI format, the design specifications can be easier understood by testing software.

7.2 Future Work

With any research or project there is always more that can be done to improve or build upon the propositions or developments. This is true of this research and in particular the design and development of the class-testing tool.

Main areas of future work have been identified:

It would worth extending the hybrid class testing technique to cluster testing technique.

BIBLIOGRAPHY

1. A. Coen-Porisini, F. De Paoli, C.Ghezzi, and D.Mandrioli. (1991) "Software specification via symbolic execution". IEEE Trans Software Engineering, SE-17 (9): 884-899, 1991

2. Alhir, S.S. (1998), UML in a nutshell, O'Reilly, 1998.

3. Aynur Abdurazik, Jeff Offutt (2000), "*Using UML Collaboration Diagrams for Static checking and test generation*", Proceedings of 3rd international conference on UML (UML'00), 2000.

4. Beizer, B. (1990) *Software Testing Techniques*, 2nd ed. Van Nostrand Reinhold.

5. Beizer, B. (1995) *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, John Wiley & Sons, Inc.

6. Binder, R. V. (1996a) "The FREE Approach for System Testing: Use-cases, Threads, and Relations", *Object Magazine*, 6(2), pp. 73-81.

7. Binder, R. V. (1996b) "Testing Object-Oriented Software: a Survey", *Software Testing, Verification and Reliability*, (6), pp. 125-252.

8. Chan, F.T. and Chen, T.Y. (1987) "AIDA-A Dynamic data flow anomaly detection system for Pascal programs", Software-practice and Experience, 17(3): 227-239

9. CHEN, H.Y, TSE, T.H., (1998), "In black and White: an integrated approach to class-level testing of object-oriented programs", ACM Trans on Software engineering and methodology, 1998, 7(3): 250-295

10. Chow, T.S., (1978), "Testing software design modeled by finite-state machines", IEEE Transactions on Software Engineering, SE-4, n.3, 178-186, 1978

11. Doong, R.-K. and Frankl, P.G.(1994), "The ASTOOT approach to testing object-oriented programs", ACM Trans on software engineering and Methodology, 1994, 3(2):101-130

12. Frankl, P. G. and Weyuker, E. J. (1988) "An Applicable Family of Data Flow Testing Criteria", *IEEE Transactions on Software Engineering*, 14(10), pp. 1483-1498.

13. Firesmith, D. G. (1993) "Testing Object-Oriented Software", *Proceedings of 11th International Conference on Technology of Object Oriented Language and Systems, (TOOLS USA, '93)*, New Jersey, pp. 407-426.

14. Fosdick, L.D. and Osterweil, L. J. (1976), "Data Flow analysis in software reliability", ACM Computing surveys 8(3): 305-330.

15. Gao, J. Z., Kung, D., Hsia, P., Toyoshima, Y. and Chen C. (1995) "Object State Testing for Object-Oriented Programs", *Proceedings of International Computer Software and Application Conference (COMPSAC '95)*, The IEEE Computer Society, pp. 232-238.

16. Glass, R.L. (1990), "Software Maintenance is solution-not a problem", Journal of systems and software, 11(2): 77-78.

17. Harel, D., (1987), "Statecharts: a visual formalism for complex systems", Science of computer programming, 1987, Vol8, 231-274

18. Harrold, M. J. and Rothermel, G. (1994) "Performing Data Flow Testing on Classes", *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 154-163.

19. Hoffman, D. and Strooper, P. (1997), "ClassBench: a Framework for Automated Class Testing", *Software - Practice and Experience*, 27(5), pp. 573-597.

20. Hong, H. S., Kwon, Y. R. and Cha, S. D. (1995), "Testing of Object-Oriented Programs Based on Finite State Machines", *Proceedings of APSEC '95*, Australia, pp. 234-241.

21. Huang, C-M., Lin, Y-C., and Jang, M-Y. (1995), "An Executable Protocol Test Sequence Generation Method for EFSM-specified Protocols", *IFIP Transactions C: Communication Systems – Protocol Test Systems*, pp 29-44.

22. Huang, J. C. (1979), "Detection of Data Flow Anomaly Through Program Instrumentation", *IEEE Transaction of Software Engineering*, SE-5 (3), pp. 226-236.

23. Jorgensen, P.C. and Erickson, C., (1994),"Object-oriented integration testing", Communications of ACM, 1994, 37(9): 30-38

24. Jeff Offutt and Aynur Abdurazik, (1999), "Generating tests from UML specifications", proceedings of the second IEEE international conference on the Unified Modeling Language (UML99), 1999, PP416-429

25. Kim, Y.G., H.S.Hong, S.M.Cho (1999), "Test cases generation from UML State Diagrams", IEE Proceedings, Software, 1999, 146(4): 187-192

26. Kirani, S and Tsai, W. T. (1994), "Method sequence specification and verification of classes", *Journal of Object-Oriented Programming*, October, pp. 28-38.

27. Kung, D. C., Gao, J., Hsia, P., Lin, J. and Toyoshima, Y. (1993), "Design Recovery for Software Testing of Object-Oriented Program", *Proceedings of the Working Conference on everse Engineering*, Baltimore Maryland, IEEE Computer Society Press, pp. 202-211.

28. Kung, D. C., Suchak, N., Gao, J. and Hsia, P. (1994), "On Object State Testing", *Proceedings of the 18th Annual International Computer Software & Applications Conference*, IEEE Computer Society Press, pp. 222-227.

29. Kung, D., Gao, J., Hsia, P., Toyoshima, Y., and Chen, C. (1995), "A Test Strategy for Object-Oriented Programs", *Proceedings of Computer Software and Applications Conference*, Dallas Texas, pp. 239-244.

30. Kung, D., Lu, N., Venugopalan, N., Hsia, P., Toyoshima, Y. Chen, C. and Gao, J (1996), " Object State Testing and Fault Analysis for Reliable Software Systems", *Proceedings of the 7th International Symposium on Software Reliability Engineering*, New York, pp. 76-85.

31. Lorenz, M. (1993) *Object-Oriented Software Development*, Englewood Cliffs, New Jersey: Prentice Hall.

32. Luo, G., Bochmann, G. v. and Petrenko, A. (1994), "Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method", IEEE Transactions on Software Engineering, 20(2), pp 149-162.

33. McGregor, J. D. and Korson, T. D. (1994), "Integrated Object-Oriented Testing and Development Processes", *Communications of the ACM*, 37(9) pp. 59-77.

34. Parrish, A. S., Borie, R. B. and Cordes, D. W. (1993), "Automated Flow Graph-Based Testing of Object-Oriented Software Modules", *Journal of Systems and Software*, 23, pp. 95-109.

35. Rapps, S. and Weyuker, E. J. (1985), "Selecting Software Test Data Using Data Flow Information", *IEEE Transactions on Software Engineering*, SE-11 (4), pp. 367-375.

36. R. M. Hierons, "Testing from a Z Specification". *Software Testing, Verification and Reliability*, Vol. 7. (1997) 19-33.

37. Tsai, B-Y., Stobart, S., Parrington, N. and Mitchell, I. (1999) "Automated Class Testing Using Threaded Multi-way Trees to Represent the Behavior of State Machines". *Volume 8, the Annals of Software Engineering Journal*, pp. 203-221.

38. Tse, T. H. and Xu, Z. (1996), "Test Case Generation for Class-Level Object-Oriented Testing", *Proceedings of the 9th international Software Quality Week (QW '96)*, San Francisco.

39. Turner, C.D. and Robson, D.J. (1995), "A state-based approach to the testing of class-based programs". Software: Concepts and Tools, 1995, 16(3): 106-112.

40. Ugo Buy, Carlo Ghezz, Alessandro Orso, Mauro Pezze, "A framework for testing object-oriented components", ICSE'99 Workshop on Testing distributed Component based Systems, 1999

41. Weyuker, E., T. Goradia and A Singh, "Automatically Generating Test Data from a Boolean pecification". *IEEE Trans on Software Engineering*, Vol. 20, No. 5. (1994), pp. 353-363.

42. XMI Specification Document, ftp://ftp.omg.org/pub/docs/ad/98-07-01.pdf

43. JavaCC web site: http://www.metamata.com/javacc

44. Novosoft's NS UML web site: http://nsuml.sourceforge.net/

Appendix A XMI file for queue class

```
<?xml version="1.0" encoding="UTF-8"?>
<XMI xmi.version="1.0">
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>Novosoft UML Library</XMI.exporter>
      <XMI.exporterVersion>0.4.19</XMI.exporterVersion>
    </XMI.documentation>
    <XMI.metamodel xmi.name="UML" xmi.version="1.3"/>
  </XMI.header>
<XMI.content>
    <Model_Management.Model xmi.id="xmi.1" xmi.uuid="127-0-0-1--6fcbeb23:e588a14:-8000">
     <Foundation.Core.ModelElement.name>untitledModel</Foundation.Core.ModelElement.name>
      <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
      <Foundation.Core.GeneralizableElement.isRoot xmi.value="false"/>
      <Foundation.Core.GeneralizableElement.isLeaf xmi.value="false"/>
      <Foundation.Core.GeneralizableElement.isAbstract xmi.value="false"/>
      <Foundation.Core.Namespace.ownedElement>
        <Foundation.Core.Class xmi.id="xmi.2" xmi.uuid="127-0-0-1-b23:e52e988a14:-7fff">
          <Foundation.Core.ModelElement.name>queue</Foundation.Core.ModelElement.name>
          <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isRoot xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isLeaf xmi.value="false"/>
          <Foundation.Core.GeneralizableElement.isAbstract xmi.value="false"/>
          <Foundation.Core.Class.isActive xmi.value="false"/>
          <Foundation.Core.ModelElement.namespace>
            <Foundation.Core.Namespace xmi.idref="xmi.1"/>
          </Foundation.Core.ModelElement.namespace>
          <Foundation.Core.Namespace.ownedElement>
          <Behavioral_Elements.State_Machines.StateMachine xmi.id="xmi.3"xmi.uuid="12">
 Foundation.Core.ModelElement.name>queueStateMachine</Foundation.Core.ModelElement.name>
              <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
              <Foundation.Core.ModelElement.namespace>
                <Foundation.Core.Namespace xmi.idref="xmi.2"/>
              </Foundation.Core.ModelElement.namespace>
              <Behavioral_Elements.State_Machines.StateMachine.context>
                <Foundation.Core.ModelElement xmi.idref="xmi.2"/>
              </Behavioral_Elements.State_Machines.StateMachine.context>
              <Behavioral_Elements.State_Machines.StateMachine.top>
                <Behavioral_Elements.State_Machines.CompositeState xmi.id="xmi.4">
 <Foundation.Core.ModelElement.name>state_machine_top</Foundation.Core.ModelElement.name>
                  <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                  <Behavioral_Elements.State_Machines.State.stateMachine>
                    <Behavioral_Elements.State_Machines.StateMachine xmi.idref="xmi.3"/>
                  </Behavioral_Elements.State_Machines.State.stateMachine>
                  <Behavioral_Elements.State_Machines.CompositeState.subvertex>
                    <Behavioral_Elements.State_Machines.Pseudostate xmi.id="xmi.5"
xmi.uuid="127-0-0-1--6fcbeb23:e52e988a14:-7ffc">
                      <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                <Behavioral_Elements.State_Machines.Pseudostate.kind xmi.value="initial"/>
                      <Behavioral_Elements.State_Machines.StateVertex.container>
                    <Behavioral_Elements.State_Machines.CompositeState xmi.idref="xmi.4"/>
                      </Behavioral_Elements.State_Machines.StateVertex.container>
                      <Behavioral_Elements.State_Machines.StateVertex.outgoing>
                      <Behavioral_Elements.State_Machines.Transition xmi.idref="xmi.6"/>
                      </Behavioral_Elements.State_Machines.StateVertex.outgoing>
                    </Behavioral_Elements.State_Machines.Pseudostate>
                    <Behavioral_Elements.State_Machines.State xmi.id="xmi.7"
xmi.uuid="127-0-0-1--6fcbeb23:e52e988a14:-7ffb">
<Foundation.Core.ModelElement.name>Empty</Foundation.Core.ModelElement.name>
                      <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                      <Behavioral_Elements.State_Machines.StateVertex.container>
                        <Behavioral_Elements.State_Machines.CompositeState
xmi.idref="xmi.4"/>
                      </Behavioral_Elements.State_Machines.StateVertex.container>
                      <Behavioral_Elements.State_Machines.StateVertex.outgoing>
```

```
                        <Behavioral_Elements.State_Machines.Transition
xmi.idref="xmi.8"/>
                      </Behavioral_Elements.State_Machines.StateVertex.outgoing>
                      <Behavioral_Elements.State_Machines.StateVertex.incoming>
                        <Behavioral_Elements.State_Machines.Transition
xmi.idref="xmi.6"/>
                        <Behavioral_Elements.State_Machines.Transition
xmi.idref="xmi.9"/>
                      </Behavioral_Elements.State_Machines.StateVertex.incoming>
                    </Behavioral_Elements.State_Machines.State>
                    <Behavioral_Elements.State_Machines.State xmi.id="xmi.10"
xmi.uuid="127-0-0-1--6fcbeb23:e52e988a14:-7ff9">
<Foundation.Core.ModelElement.name>NotFull</Foundation.Core.ModelElement.name>
                      <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                      <Behavioral_Elements.State_Machines.StateVertex.container>
                        <Behavioral_Elements.State_Machines.CompositeState
xmi.idref="xmi.4"/>
                      </Behavioral_Elements.State_Machines.StateVertex.container>
                      <Behavioral_Elements.State_Machines.StateVertex.outgoing>
                        <Behavioral_Elements.State_Machines.Transition
xmi.idref="xmi.9"/>
                        <Behavioral_Elements.State_Machines.Transition
xmi.idref="xmi.11"/>
                      </Behavioral_Elements.State_Machines.StateVertex.outgoing>
                      <Behavioral_Elements.State_Machines.StateVertex.incoming>
                        <Behavioral_Elements.State_Machines.Transition
xmi.idref="xmi.8"/>
                        <Behavioral_Elements.State_Machines.Transition
xmi.idref="xmi.12"/>
                      </Behavioral_Elements.State_Machines.StateVertex.incoming>
                    </Behavioral_Elements.State_Machines.State>
                    <Behavioral_Elements.State_Machines.State xmi.id="xmi.13"
xmi.uuid="127-0-0-1--6fcbeb23:e52e988a14:-7ff8">

<Foundation.Core.ModelElement.name>Full</Foundation.Core.ModelElement.name>
                      <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                      <Behavioral_Elements.State_Machines.StateVertex.container>
                        <Behavioral_Elements.State_Machines.CompositeState
xmi.idref="xmi.4"/>
                      </Behavioral_Elements.State_Machines.StateVertex.container>
                      <Behavioral_Elements.State_Machines.StateVertex.outgoing>
                        <Behavioral_Elements.State_Machines.Transition
xmi.idref="xmi.12"/>
                      </Behavioral_Elements.State_Machines.StateVertex.outgoing>
                      <Behavioral_Elements.State_Machines.StateVertex.incoming>
                        <Behavioral_Elements.State_Machines.Transition
xmi.idref="xmi.11"/>
                      </Behavioral_Elements.State_Machines.StateVertex.incoming>
                    </Behavioral_Elements.State_Machines.State>
                  </Behavioral_Elements.State_Machines.CompositeState.subvertex>
                </Behavioral_Elements.State_Machines.CompositeState>
              </Behavioral_Elements.State_Machines.StateMachine.top>
              <Behavioral_Elements.State_Machines.StateMachine.transitions>
                <Behavioral_Elements.State_Machines.Transition xmi.id="xmi.6"
xmi.uuid="127-0-0-1--6fcbeb23:e52e988a14:-7ffa">
                  <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                  <Behavioral_Elements.State_Machines.Transition.stateMachine>
                    <Behavioral_Elements.State_Machines.StateMachine xmi.idref="xmi.3"/>
                  </Behavioral_Elements.State_Machines.Transition.stateMachine>
                  <Behavioral_Elements.State_Machines.Transition.source>
                    <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.5"/>
                  </Behavioral_Elements.State_Machines.Transition.source>
                  <Behavioral_Elements.State_Machines.Transition.target>
                    <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.7"/>
                  </Behavioral_Elements.State_Machines.Transition.target>
                </Behavioral_Elements.State_Machines.Transition>
                <Behavioral_Elements.State_Machines.Transition xmi.id="xmi.8"
xmi.uuid="127-0-0-1--6fcbeb23:e52e988a14:-7ff7">
<Foundation.Core.ModelElement.name>add</Foundation.Core.ModelElement.name>
                  <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                  <Behavioral_Elements.State_Machines.Transition.trigger>
```

```
                    <Behavioral_Elements.State_Machines.Event xmi.idref="xmi.14"/>
                  </Behavioral_Elements.State_Machines.Transition.trigger>
                  <Behavioral_Elements.State_Machines.Transition.stateMachine>
                    <Behavioral_Elements.State_Machines.StateMachine xmi.idref="xmi.3"/>
                  </Behavioral_Elements.State_Machines.Transition.stateMachine>
                  <Behavioral_Elements.State_Machines.Transition.source>
                    <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.7"/>
                  </Behavioral_Elements.State_Machines.Transition.source>
                  <Behavioral_Elements.State_Machines.Transition.target>
                    <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.10"/>
                  </Behavioral_Elements.State_Machines.Transition.target>
                  <Behavioral_Elements.State_Machines.Transition.guard>
                    <Behavioral_Elements.State_Machines.Guard xmi.id="xmi.15">
                      <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                      <Behavioral_Elements.State_Machines.Guard.expression>
                        <Foundation.Data_Types.BooleanExpression xmi.id="xmi.16">
<Foundation.Data_Types.Expression.language>bool</Foundation.Data_Types.Expression.language>
<Foundation.Data_Types.Expression.body>count&lt;5</Foundation.Data_Types.Expression.body>
                        </Foundation.Data_Types.BooleanExpression>
                      </Behavioral_Elements.State_Machines.Guard.expression>
                      <Behavioral_Elements.State_Machines.Guard.transition>
                        <Behavioral_Elements.State_Machines.Transition
xmi.idref="xmi.8"/>
                      </Behavioral_Elements.State_Machines.Guard.transition>
                    </Behavioral_Elements.State_Machines.Guard>
                  </Behavioral_Elements.State_Machines.Transition.guard>
                  <Behavioral_Elements.State_Machines.Transition.effect>
                    <Behavioral_Elements.Common_Behavior.ActionSequence xmi.id="xmi.17">
<Foundation.Core.ModelElement.name></Foundation.Core.ModelElement.name>
                      <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                      <Behavioral_Elements.Common_Behavior.Action.isAsynchronous
xmi.value="false"/>
                    </Behavioral_Elements.Common_Behavior.ActionSequence>
                  </Behavioral_Elements.State_Machines.Transition.effect>
                </Behavioral_Elements.State_Machines.Transition>
                <Behavioral_Elements.State_Machines.Transition xmi.id="xmi.9"
xmi.uuid="127-0-0-1--6fcbeb23:e52e988a14:-7ff6">
<Foundation.Core.ModelElement.name>delet</Foundation.Core.ModelElement.name>
                  <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                  <Behavioral_Elements.State_Machines.Transition.trigger>
                    <Behavioral_Elements.State_Machines.Event xmi.idref="xmi.18"/>
                  </Behavioral_Elements.State_Machines.Transition.trigger>
                  <Behavioral_Elements.State_Machines.Transition.stateMachine>
                    <Behavioral_Elements.State_Machines.StateMachine xmi.idref="xmi.3"/>
                  </Behavioral_Elements.State_Machines.Transition.stateMachine>
                  <Behavioral_Elements.State_Machines.Transition.source>
                    <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.10"/>
                  </Behavioral_Elements.State_Machines.Transition.source>
                  <Behavioral_Elements.State_Machines.Transition.target>
                    <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.7"/>
                  </Behavioral_Elements.State_Machines.Transition.target>
                  <Behavioral_Elements.State_Machines.Transition.guard>
                    <Behavioral_Elements.State_Machines.Guard xmi.id="xmi.19">
                      <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                      <Behavioral_Elements.State_Machines.Guard.expression>
                        <Foundation.Data_Types.BooleanExpression xmi.id="xmi.20">
<Foundation.Data_Types.Expression.language>bool</Foundation.Data_Types.Expression.languag
e>
<Foundation.Data_Types.Expression.body>count&gt;0</Foundation.Data_Types.Expression.body>
                        </Foundation.Data_Types.BooleanExpression>
                      </Behavioral_Elements.State_Machines.Guard.expression>
                      <Behavioral_Elements.State_Machines.Guard.transition>
                       <Behavioral_Elements.State_Machines.Transition xmi.idref="xmi.9"/>
                      </Behavioral_Elements.State_Machines.Guard.transition>
                    </Behavioral_Elements.State_Machines.Guard>
                  </Behavioral_Elements.State_Machines.Transition.guard>
                </Behavioral_Elements.State_Machines.Transition>
                <Behavioral_Elements.State_Machines.Transition xmi.id="xmi.11"
xmi.uuid="127-0-0-1--6fcbeb23:e52e988a14:-7ff5">
<Foundation.Core.ModelElement.name>add</Foundation.Core.ModelElement.name>
                  <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
```

```
                    <Behavioral_Elements.State_Machines.Transition.trigger>
                      <Behavioral_Elements.State_Machines.Event xmi.idref="xmi.21"/>
                    </Behavioral_Elements.State_Machines.Transition.trigger>
                    <Behavioral_Elements.State_Machines.Transition.stateMachine>
                      <Behavioral_Elements.State_Machines.StateMachine xmi.idref="xmi.3"/>
                    </Behavioral_Elements.State_Machines.Transition.stateMachine>
                    <Behavioral_Elements.State_Machines.Transition.source>
                      <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.10"/>
                    </Behavioral_Elements.State_Machines.Transition.source>
                    <Behavioral_Elements.State_Machines.Transition.target>
                      <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.13"/>
                    </Behavioral_Elements.State_Machines.Transition.target>
                    <Behavioral_Elements.State_Machines.Transition.guard>
                      <Behavioral_Elements.State_Machines.Guard xmi.id="xmi.22">
                        <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                        <Behavioral_Elements.State_Machines.Guard.expression>
                          <Foundation.Data_Types.BooleanExpression xmi.id="xmi.23">
<Foundation.Data_Types.Expression.language>ocl</Foundation.Data_Types.Expression.language>
<Foundation.Data_Types.Expression.body>count&lt;5</Foundation.Data_Types.Expression.body>
                          </Foundation.Data_Types.BooleanExpression>
                        </Behavioral_Elements.State_Machines.Guard.expression>
                        <Behavioral_Elements.State_Machines.Guard.transition>
                        <Behavioral_Elements.State_Machines.Transition xmi.idref="xmi.11"/>
                        </Behavioral_Elements.State_Machines.Guard.transition>
                      </Behavioral_Elements.State_Machines.Guard>
                    </Behavioral_Elements.State_Machines.Transition.guard>
                  </Behavioral_Elements.State_Machines.Transition>
                  <Behavioral_Elements.State_Machines.Transition xmi.id="xmi.12"
xmi.uuid="127-0-0-1--6fcbeb23:e52e988a14:-7ff4">
<Foundation.Core.ModelElement.name>delete</Foundation.Core.ModelElement.name>
                    <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                    <Behavioral_Elements.State_Machines.Transition.trigger>
                      <Behavioral_Elements.State_Machines.Event xmi.idref="xmi.24"/>
                    </Behavioral_Elements.State_Machines.Transition.trigger>
                    <Behavioral_Elements.State_Machines.Transition.stateMachine>
                      <Behavioral_Elements.State_Machines.StateMachine xmi.idref="xmi.3"/>
                    </Behavioral_Elements.State_Machines.Transition.stateMachine>
                    <Behavioral_Elements.State_Machines.Transition.source>
                      <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.13"/>
                    </Behavioral_Elements.State_Machines.Transition.source>
                    <Behavioral_Elements.State_Machines.Transition.target>
                      <Behavioral_Elements.State_Machines.StateVertex xmi.idref="xmi.10"/>
                    </Behavioral_Elements.State_Machines.Transition.target>
                    <Behavioral_Elements.State_Machines.Transition.guard>
                      <Behavioral_Elements.State_Machines.Guard xmi.id="xmi.25">
                        <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                        <Behavioral_Elements.State_Machines.Guard.expression>
                          <Foundation.Data_Types.BooleanExpression xmi.id="xmi.26">
<Foundation.Data_Types.Expression.language>ocl</Foundation.Data_Types.Expression.language>
<Foundation.Data_Types.Expression.body>count&gt;0</Foundation.Data_Types.Expression.body>
                          </Foundation.Data_Types.BooleanExpression>
                        </Behavioral_Elements.State_Machines.Guard.expression>
                        <Behavioral_Elements.State_Machines.Guard.transition>
                        <Behavioral_Elements.State_Machines.Transition xmi.idref="xmi.12"/>
                        </Behavioral_Elements.State_Machines.Guard.transition>
                      </Behavioral_Elements.State_Machines.Guard>
                    </Behavioral_Elements.State_Machines.Transition.guard>
                  </Behavioral_Elements.State_Machines.Transition>
                </Behavioral_Elements.State_Machines.StateMachine.transitions>
              </Behavioral_Elements.State_Machines.StateMachine>
            </Foundation.Core.Namespace.ownedElement>
            <Foundation.Core.Classifier.feature>
              <Foundation.Core.Attribute xmi.id="xmi.27">
               <Foundation.Core.ModelElement.name>count</Foundation.Core.ModelElement.name>
                <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
                <Foundation.Core.Feature.owner>
                  <Foundation.Core.Classifier xmi.idref="xmi.2"/>
                </Foundation.Core.Feature.owner>
              </Foundation.Core.Attribute>
              <Foundation.Core.Attribute xmi.id="xmi.28">
                <Foundation.Core.ModelElement.name>Q</Foundation.Core.ModelElement.name>
```

```
        <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
        <Foundation.Core.Feature.owner>
          <Foundation.Core.Classifier xmi.idref="xmi.2"/>
        </Foundation.Core.Feature.owner>
      </Foundation.Core.Attribute>
      <Foundation.Core.Operation xmi.id="xmi.29">
       <Foundation.Core.ModelElement.name>queue</Foundation.Core.ModelElement.name>
        <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
        <Foundation.Core.BehavioralFeature.isQuery xmi.value="false"/>
        <Foundation.Core.Operation.isRoot xmi.value="false"/>
        <Foundation.Core.Operation.isLeaf xmi.value="false"/>
        <Foundation.Core.Operation.isAbstract xmi.value="false"/>
        <Foundation.Core.Feature.owner>
          <Foundation.Core.Classifier xmi.idref="xmi.2"/>
        </Foundation.Core.Feature.owner>
      </Foundation.Core.Operation>
      <Foundation.Core.Operation xmi.id="xmi.30">
        <Foundation.Core.ModelElement.name>add</Foundation.Core.ModelElement.name>
        <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
        <Foundation.Core.BehavioralFeature.isQuery xmi.value="false"/>
        <Foundation.Core.Operation.isRoot xmi.value="false"/>
        <Foundation.Core.Operation.isLeaf xmi.value="false"/>
        <Foundation.Core.Operation.isAbstract xmi.value="false"/>
        <Foundation.Core.Feature.owner>
          <Foundation.Core.Classifier xmi.idref="xmi.2"/>
        </Foundation.Core.Feature.owner>
        <Foundation.Core.BehavioralFeature.parameter>
          <Foundation.Core.Parameter xmi.id="xmi.31">
        <Foundation.Core.ModelElement.name>data</Foundation.Core.ModelElement.name>
            <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
            <Foundation.Core.Parameter.kind xmi.value="in"/>
            <Foundation.Core.Parameter.behavioralFeature>
              <Foundation.Core.BehavioralFeature xmi.idref="xmi.30"/>
            </Foundation.Core.Parameter.behavioralFeature>
            <Foundation.Core.Parameter.type>
              <Foundation.Core.Classifier xmi.idref="xmi.32"/>
            </Foundation.Core.Parameter.type>
          </Foundation.Core.Parameter>
        </Foundation.Core.BehavioralFeature.parameter>
      </Foundation.Core.Operation>
      <Foundation.Core.Operation xmi.id="xmi.33">
       <Foundation.Core.ModelElement.name>delte</Foundation.Core.ModelElement.name>
        <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
        <Foundation.Core.BehavioralFeature.isQuery xmi.value="false"/>
        <Foundation.Core.Operation.isRoot xmi.value="false"/>
        <Foundation.Core.Operation.isLeaf xmi.value="false"/>
        <Foundation.Core.Operation.isAbstract xmi.value="false"/>
        <Foundation.Core.Feature.owner>
          <Foundation.Core.Classifier xmi.idref="xmi.2"/>
        </Foundation.Core.Feature.owner>
      </Foundation.Core.Operation>
      <Foundation.Core.Operation xmi.id="xmi.34">
        <Foundation.Core.ModelElement.name>size</Foundation.Core.ModelElement.name>
        <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
        <Foundation.Core.BehavioralFeature.isQuery xmi.value="false"/>
        <Foundation.Core.Operation.isRoot xmi.value="false"/>
        <Foundation.Core.Operation.isLeaf xmi.value="false"/>
        <Foundation.Core.Operation.isAbstract xmi.value="false"/>
        <Foundation.Core.Feature.owner>
          <Foundation.Core.Classifier xmi.idref="xmi.2"/>
        </Foundation.Core.Feature.owner>
      </Foundation.Core.Operation>
    </Foundation.Core.Classifier.feature>
  </Foundation.Core.Class>
  <Foundation.Core.DataType xmi.id="xmi.32">
    <Foundation.Core.ModelElement.name>char</Foundation.Core.ModelElement.name>
    <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
    <Foundation.Core.GeneralizableElement.isRoot xmi.value="false"/>
    <Foundation.Core.GeneralizableElement.isLeaf xmi.value="false"/>
    <Foundation.Core.GeneralizableElement.isAbstract xmi.value="false"/>
    <Foundation.Core.ModelElement.namespace>
```

```
            <Foundation.Core.Namespace xmi.idref="xmi.1"/>
          </Foundation.Core.ModelElement.namespace>
        </Foundation.Core.DataType>
        <Behavioral_Elements.State_Machines.SignalEvent xmi.id="xmi.14">
          <Foundation.Core.ModelElement.name>t1</Foundation.Core.ModelElement.name>
          <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
          <Foundation.Core.ModelElement.namespace>
            <Foundation.Core.Namespace xmi.idref="xmi.1"/>
          </Foundation.Core.ModelElement.namespace>
          <Behavioral_Elements.State_Machines.Event.transition>
            <Behavioral_Elements.State_Machines.Transition xmi.idref="xmi.35"/>
            <Behavioral_Elements.State_Machines.Transition xmi.idref="xmi.8"/>
          </Behavioral_Elements.State_Machines.Event.transition>
        </Behavioral_Elements.State_Machines.SignalEvent>
        <Behavioral_Elements.State_Machines.SignalEvent xmi.id="xmi.18">
          <Foundation.Core.ModelElement.name>t2</Foundation.Core.ModelElement.name>
          <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
          <Foundation.Core.ModelElement.namespace>
            <Foundation.Core.Namespace xmi.idref="xmi.1"/>
          </Foundation.Core.ModelElement.namespace>
          <Behavioral_Elements.State_Machines.Event.transition>
            <Behavioral_Elements.State_Machines.Transition xmi.idref="xmi.9"/>
          </Behavioral_Elements.State_Machines.Event.transition>
        </Behavioral_Elements.State_Machines.SignalEvent>
        <Behavioral_Elements.State_Machines.SignalEvent xmi.id="xmi.21">
          <Foundation.Core.ModelElement.name>t3</Foundation.Core.ModelElement.name>
          <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
          <Foundation.Core.ModelElement.namespace>
            <Foundation.Core.Namespace xmi.idref="xmi.1"/>
          </Foundation.Core.ModelElement.namespace>
          <Behavioral_Elements.State_Machines.Event.transition>
            <Behavioral_Elements.State_Machines.Transition xmi.idref="xmi.11"/>
          </Behavioral_Elements.State_Machines.Event.transition>
        </Behavioral_Elements.State_Machines.SignalEvent>
        <Behavioral_Elements.State_Machines.SignalEvent xmi.id="xmi.24">
          <Foundation.Core.ModelElement.name>t4</Foundation.Core.ModelElement.name>
          <Foundation.Core.ModelElement.isSpecification xmi.value="false"/>
          <Foundation.Core.ModelElement.namespace>
            <Foundation.Core.Namespace xmi.idref="xmi.1"/>
          </Foundation.Core.ModelElement.namespace>
          <Behavioral_Elements.State_Machines.Event.transition>
            <Behavioral_Elements.State_Machines.Transition xmi.idref="xmi.12"/>
          </Behavioral_Elements.State_Machines.Event.transition>
        </Behavioral_Elements.State_Machines.SignalEvent>
      </Foundation.Core.Namespace.ownedElement>
    </Model_Management.Model>
  </XMI.content>
</XMI>
```

Appendix B

```
//Source Code Parser Grammar File
 *
 *
 * Author: Xiaohong Yang
 * Date:   Feb. 2001
 *
 * This file contains a Java grammar and
actions that implement a front-end.
 *
 */

options {
  JAVA_UNICODE_ESCAPE = true;
}

PARSER_BEGIN(JavaParser)

public class JavaParser {

  public static void main(String args[]) {
    JavaParser parser;
    if (args.length == 0) {
      System.out.println("Java Parser
Version 1.1:  Reading from standard input .
. .");
      parser = new JavaParser(System.in);
    } else if (args.length == 1) {
      System.out.println("Java Parser
Version 1.1:  Reading from file " + args[0]
+ " . . .");
      try {
        parser = new JavaParser(new
java.io.FileInputStream(args[0]));
      } catch
(java.io.FileNotFoundException e) {
        System.out.println("Java Parser
Version 1.1:  File " + args[0] + " not
found.");
        return;
      }
    } else {
      System.out.println("Java Parser
Version 1.1:  Usage is one of:");
      System.out.println("         java
JavaParser < inputfile");
      System.out.println("OR");
      System.out.println("         java
JavaParser inputfile");
      return;
    }
    try {
      parser.CompilationUnit();
      System.out.println("Java Parser
Version 1.1:  Java program parsed
successfully.");
    } catch (ParseError e) {
      System.out.println(e.getMessage());
      System.out.println("Java Parser
Version 1.1:  Encountered errors during
parse.");
    }
  }

}

PARSER_END(JavaParser)

/* WHITE SPACE */

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}

/* COMMENTS */

MORE :
{
  "//" : IN_SINGLE_LINE_COMMENT
|
  <"/**" ~["/"]> : IN_FORMAL_COMMENT
|
  "/*" : IN_MULTI_LINE_COMMENT
}

<IN_SINGLE_LINE_COMMENT>
SPECIAL_TOKEN :
{
  <SINGLE_LINE_COMMENT: "\n" | "\r" |
"\r\n" > : DEFAULT
}

<IN_FORMAL_COMMENT>
SPECIAL_TOKEN :
{
  <FORMAL_COMMENT: "*/" > : DEFAULT
}

<IN_MULTI_LINE_COMMENT>
SPECIAL_TOKEN :
{
  <MULTI_LINE_COMMENT: "*/" > : DEFAULT
}

<IN_SINGLE_LINE_COMMENT,IN_FORMAL_COMMENT,I
N_MULTI_LINE_COMMENT>
MORE :
{
  < ~[] >
}

/* RESERVED WORDS AND LITERALS */

TOKEN :
{
  < ABSTRACT: "abstract" >
| < BOOLEAN: "boolean" >
| < BREAK: "break" >
| < BYTE: "byte" >
| < CASE: "case" >
| < CATCH: "catch" >
| < CHAR: "char" >
| < CLASS: "class" >
| < CONST: "const" >
| < CONTINUE: "continue" >
| < _DEFAULT: "default" >
| < DO: "do" >
| < DOUBLE: "double" >
| < ELSE: "else" >
| < EXTENDS: "extends" >
| < FALSE: "false" >
| < FINAL: "final" >
```

```
| < FINALLY: "finally" >
| < FLOAT: "float" >
| < FOR: "for" >
| < GOTO: "goto" >
| < IF: "if" >
| < IMPLEMENTS: "implements" >
| < IMPORT: "import" >
| < INSTANCEOF: "instanceof" >
| < INT: "int" >
| < INTERFACE: "interface" >
| < LONG: "long" >
| < NATIVE: "native" >
| < NEW: "new" >
| < NULL: "null" >
| < PACKAGE: "package">
| < PRIVATE: "private" >
| < PROTECTED: "protected" >
| < PUBLIC: "public" >
| < RETURN: "return" >
| < SHORT: "short" >
| < STATIC: "static" >
| < SUPER: "super" >
| < SWITCH: "switch" >
| < SYNCHRONIZED: "synchronized" >
| < THIS: "this" >
| < THROW: "throw" >
| < THROWS: "throws" >
| < TRANSIENT: "transient" >
| < TRUE: "true" >
| < TRY: "try" >
| < VOID: "void" >
| < VOLATILE: "volatile" >
| < WHILE: "while" >
}

/* LITERALS */

TOKEN :
{
  < INTEGER_LITERAL:
        <DECIMAL_LITERAL> (["l","L"])?
      | <HEX_LITERAL> (["l","L"])?
      | <OCTAL_LITERAL> (["l","L"])?
  >
|
  < #DECIMAL_LITERAL: ["1"-"9"] (["0"-
"9"])* >
|
  < #HEX_LITERAL: "0" ["x","X"] (["0"-
"9","a"-"f","A"-"F"])+ >
|
  < #OCTAL_LITERAL: "0" (["0"-"7"])* >
|
  < FLOATING_POINT_LITERAL:
        (["0"-"9"])+ "." (["0"-"9"])*
(<EXPONENT>)? (["f","F","d","D"])?
      | "." (["0"-"9"])+ (<EXPONENT>)?
(["f","F","d","D"])?
      | (["0"-"9"])+ <EXPONENT>
(["f","F","d","D"])?
      | (["0"-"9"])+ (<EXPONENT>)?
["f","F","d","D"]
  >
|
  < #EXPONENT: ["e","E"] (["+","-"])?
(["0"-"9"])+ >
|
  < CHARACTER_LITERAL:
      "'"
      (   (~["'","\\","\n","\r"])
        | ("\\"
            (
["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )
      "'"
  >
|
  < STRING_LITERAL:
      "\""
      (   (~["\"","\\","\n","\r"])
        | ("\\"
            (
["n","t","b","r","f","\\","'","\""]
            | ["0"-"7"] ( ["0"-"7"] )?
            | ["0"-"3"] ["0"-"7"] ["0"-"7"]
            )
          )
      )*
      "\""
  >
}

/* IDENTIFIERS */

TOKEN :
{
  < IDENTIFIER: <LETTER>
(<LETTER>|<DIGIT>)* >
|
  < #LETTER:
      [
       "\u0024",
       "\u0041"-"\u005a",
       "\u005f",
       "\u0061"-"\u007a",
       "\u00c0"-"\u00d6",
       "\u00d8"-"\u00f6",
       "\u00f8"-"\u00ff",
       "\u0100"-"\u1fff",
       "\u3040"-"\u318f",
       "\u3300"-"\u337f",
       "\u3400"-"\u3d2d",
       "\u4e00"-"\u9fff",
       "\uf900"-"\ufaff"
      ]
  >
|
  < #DIGIT:
      [
       "\u0030"-"\u0039",
       "\u0660"-"\u0669",
       "\u06f0"-"\u06f9",
       "\u0966"-"\u096f",
       "\u09e6"-"\u09ef",
       "\u0a66"-"\u0a6f",
       "\u0ae6"-"\u0aef",
       "\u0b66"-"\u0b6f",
       "\u0be7"-"\u0bef",
       "\u0c66"-"\u0c6f",
       "\u0ce6"-"\u0cef",
       "\u0d66"-"\u0d6f",
       "\u0e50"-"\u0e59",
       "\u0ed0"-"\u0ed9",
       "\u1040"-"\u1049"
      ]
  >
}
```

```
/* SEPARATORS */

TOKEN :
{
  < LPAREN: "(" >
| < RPAREN: ")" >
| < LBRACE: "{" >
| < RBRACE: "}" >
| < LBRACKET: "[" >
| < RBRACKET: "]" >
| < SEMICOLON: ";" >
| < COMMA: "," >
| < DOT: "." >
}

/* OPERATORS */

TOKEN :
{
  < ASSIGN: "=" >
| < GT: ">" >
| < LT: "<" >
| < BANG: "!" >
| < TILDE: "~" >
| < HOOK: "?" >
| < COLON: ":" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NE: "!=" >
| < SC_OR: "||" >
| < SC_AND: "&&" >
| < INCR: "++" >
| < DECR: "--" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < BIT_AND: "&" >
| < BIT_OR: "|" >
| < XOR: "^" >
| < REM: "%" >
| < LSHIFT: "<<" >
| < RSIGNEDSHIFT: ">>" >
| < RUNSIGNEDSHIFT: ">>>" >
| < PLUSASSIGN: "+=" >
| < MINUSASSIGN: "-=" >
| < STARASSIGN: "*=" >
| < SLASHASSIGN: "/=" >
| < ANDASSIGN: "&=" >
| < ORASSIGN: "|=" >
| < XORASSIGN: "^=" >
| < REMASSIGN: "%=" >
| < LSHIFTASSIGN: "<<=" >
| < RSIGNEDSHIFTASSIGN: ">>=" >
| < RUNSIGNEDSHIFTASSIGN: ">>>=" >
}


/*****************************************
 * THE JAVA LANGUAGE GRAMMAR STARTS HERE *
 *****************************************/

/*
 * Program structuring syntax follows.
 */

void CompilationUnit() :
{}
{
  [ PackageDeclaration() ]
  ( ImportDeclaration() )*
  ( TypeDeclaration() )*
  <EOF>
}

void PackageDeclaration() :
{}
{
  "package" Name() ";"
}

void ImportDeclaration() :
{}
{
  "import" Name() [ "." "*" ] ";"
}

void TypeDeclaration() :
{}
{
  LOOKAHEAD( ( "abstract" | "final" |
"public" )* "class" )
  ClassDeclaration()
|
  InterfaceDeclaration()
|
  ";"
}


/*
 * Declaration syntax follows.
 */

void ClassDeclaration() :
{}
{
  ( "abstract" | "final" | "public" )*
  UnmodifiedClassDeclaration()
}

void UnmodifiedClassDeclaration() :
{}
{
  "class" <IDENTIFIER> [ "extends" Name() ]
[ "implements" NameList() ]
  ClassBody()
}

void ClassBody() :
{}
{
  "{" ( ClassBodyDeclaration() )* "}"
}

void NestedClassDeclaration() :
{}
{
  ( "static" | "abstract" | "final" |
"public" | "protected" | "private" )*
  UnmodifiedClassDeclaration()
}

void ClassBodyDeclaration() :
{}
{
  LOOKAHEAD(2)
  Initializer()
|
```

```
  LOOKAHEAD( ( "static" | "abstract" |
"final" | "public" | "protected" |
"private" )* "class" )
  NestedClassDeclaration()
|
  LOOKAHEAD( ( "static" | "abstract" |
"final" | "public" | "protected" |
"private" )* "interface" )
  NestedInterfaceDeclaration()
|
  LOOKAHEAD( [ "public" | "protected" |
"private" ] Name() "(" )
  ConstructorDeclaration()
|
  LOOKAHEAD( MethodDeclarationLookahead() )
  MethodDeclaration()
|
  FieldDeclaration()
}

// This production is to determine
lookahead only.
void MethodDeclarationLookahead() :
{}
{
  ( "public" | "protected" | "private" |
"static" | "abstract" | "final" | "native"
| "synchronized" )*
  ResultType() <IDENTIFIER> "("
}

void InterfaceDeclaration() :
{}
{
  ( "abstract" | "public" )*
  UnmodifiedInterfaceDeclaration()
}

void NestedInterfaceDeclaration() :
{}
{
  ( "static" | "abstract" | "final" |
"public" | "protected" | "private" )*
  UnmodifiedInterfaceDeclaration()
}

void UnmodifiedInterfaceDeclaration() :
{}
{
  "interface" <IDENTIFIER> [ "extends"
NameList() ]
  "{" ( InterfaceMemberDeclaration() )* "}"
}

void InterfaceMemberDeclaration() :
{}
{
  LOOKAHEAD( ( "static" | "abstract" |
"final" | "public" | "protected" |
"private" )* "class" )
  NestedClassDeclaration()
|
  LOOKAHEAD( ( "static" | "abstract" |
"final" | "public" | "protected" |
"private" )* "interface" )
  NestedInterfaceDeclaration()
|
  LOOKAHEAD( MethodDeclarationLookahead() )
  MethodDeclaration()
|
  FieldDeclaration()
}

void FieldDeclaration() :
{}
{
  ( "public" | "protected" | "private" |
"static" | "final" | "transient" |
"volatile" )*
  Type() VariableDeclarator() ( ","
VariableDeclarator() )* ";"
}

void VariableDeclarator() :
{}
{
  VariableDeclaratorId() [ "="
VariableInitializer() ]
}

void VariableDeclaratorId() :
{}
{
  <IDENTIFIER> ( "[" "]" )*
}

void VariableInitializer() :
{}
{
  ArrayInitializer()
|
  Expression()
}

void ArrayInitializer() :
{}
{
  "{" [ VariableInitializer() (
LOOKAHEAD(2) "," VariableInitializer() )* ]
[ "," ] "}"
}

void MethodDeclaration() :
{}
{
  ( "public" | "protected" | "private" |
"static" | "abstract" | "final" | "native"
| "synchronized" )*
  ResultType() MethodDeclarator() [
"throws" NameList() ]
  ( Block() | ";" )
}

void MethodDeclarator() :
{}
{
  <IDENTIFIER> FormalParameters() ( "[" "]"
)*
}

void FormalParameters() :
{}
{
  "(" [ FormalParameter() ( ","
FormalParameter() )* ] ")"
}

void FormalParameter() :
{}
{
  [ "final" ] Type() VariableDeclaratorId()
}
```

```
void ConstructorDeclaration() :
{}
{
  [ "public" | "protected" | "private" ]
  <IDENTIFIER> FormalParameters() [
"throws" NameList() ]
  "{"
    [
LOOKAHEAD(ExplicitConstructorInvocation())
ExplicitConstructorInvocation() ]
    ( BlockStatement() )*
  "}"
}

void ExplicitConstructorInvocation() :
{}
{
  LOOKAHEAD("this" Arguments() ";")
  "this" Arguments() ";"
|
  [ LOOKAHEAD(2) PrimaryExpression() "." ]
"super" Arguments() ";"
}

void Initializer() :
{}
{
  [ "static" ] Block()
}


/*
 * Type, name and expression syntax
follows.
 */

void Type() :
{}
{
  ( PrimitiveType() | Name() ) ( "[" "]" )*
}

void PrimitiveType() :
{}
{
  "boolean"
|
  "char"
|
  "byte"
|
  "short"
|
  "int"
|
  "long"
|
  "float"
|
  "double"
}

void ResultType() :
{}
{
  "void"
|
  Type()
}

void Name() :
/*
 * A lookahead of 2 is required below since
"Name" can be followed
 * by a ".*" when used in the context of an
"ImportDeclaration".
 */
{}
{
  <IDENTIFIER>
  ( LOOKAHEAD(2) "." <IDENTIFIER>
  )*
}

void NameList() :
{}
{
  Name()
  ( "," Name()
  )*
}


/*
 * Expression syntax follows.
 */

void Expression() :
/*
 * This expansion has been written this way
instead of:
 *   Assignment() | ConditionalExpression()
 * for performance reasons.
 * However, it is a weakening of the
grammar for it allows the LHS of
 * assignments to be any conditional
expression whereas it can only be
 * a primary expression.  Consider adding a
semantic predicate to work
 * around this.
 */
{}
{
  ConditionalExpression()
  [
    AssignmentOperator() Expression()
  ]
}

void AssignmentOperator() :
{}
{
  "=" | "*=" | "/=" | "%=" | "+=" | "-=" |
"<<=" | ">>=" | ">>>=" | "&=" | "^=" | "|="
}

void ConditionalExpression() :
{}
{
  ConditionalOrExpression() [ "?"
Expression() ":" ConditionalExpression() ]
}

void ConditionalOrExpression() :
{}
{
  ConditionalAndExpression() ( "||"
ConditionalAndExpression() )*
}

void ConditionalAndExpression() :
```

```
{}
{
  InclusiveOrExpression() ( "&&"
InclusiveOrExpression() )*
}

void InclusiveOrExpression() :
{}
{
  ExclusiveOrExpression() ( "|"
ExclusiveOrExpression() )*
}

void ExclusiveOrExpression() :
{}
{
  AndExpression() ( "^" AndExpression() )*
}

void AndExpression() :
{}
{
  EqualityExpression() ( "&"
EqualityExpression() )*
}

void EqualityExpression() :
{}
{
  InstanceOfExpression() ( ( "==" | "!=" )
InstanceOfExpression() )*
}

void InstanceOfExpression() :
{}
{
  RelationalExpression() [ "instanceof"
Type() ]
}

void RelationalExpression() :
{}
{
  ShiftExpression() ( ( "<" | ">" | "<=" |
">=" ) ShiftExpression() )*
}

void ShiftExpression() :
{}
{
  AdditiveExpression() ( ( "<<" | ">>" |
">>>" ) AdditiveExpression() )*
}

void AdditiveExpression() :
{}
{
  MultiplicativeExpression() ( ( "+" | "-"
) MultiplicativeExpression() )*
}

void MultiplicativeExpression() :
{}
{
  UnaryExpression() ( ( "*" | "/" | "%" )
UnaryExpression() )*
}

void UnaryExpression() :
{}
{
  ( "+" | "-" ) UnaryExpression()
|
  PreIncrementExpression()
|
  PreDecrementExpression()
|
  UnaryExpressionNotPlusMinus()
}

void PreIncrementExpression() :
{}
{
  "++" PrimaryExpression()
}

void PreDecrementExpression() :
{}
{
  "--" PrimaryExpression()
}

void UnaryExpressionNotPlusMinus() :
{}
{
  ( "~" | "!" ) UnaryExpression()
|
  LOOKAHEAD( CastLookahead() )
  CastExpression()
|
  PostfixExpression()
}

// This production is to determine
lookahead only.  The LOOKAHEAD
specifications
// below are not used, but they are there
just to indicate that we know about
// this.
void CastLookahead() :
{}
{
  LOOKAHEAD(2)
  "(" PrimitiveType()
|
  LOOKAHEAD("(" Name() "[")
  "(" Name() "[" "]"
|
  "(" Name() ")" ( "~" | "!" | "(" |
<IDENTIFIER> | "this" | "super" | "new" |
Literal() )
}

void PostfixExpression() :
{}
{
  PrimaryExpression() [ "++" | "--" ]
}

void CastExpression() :
{}
{
  LOOKAHEAD("(" PrimitiveType())
  "(" Type() ")" UnaryExpression()
|
  LOOKAHEAD("(" Name())
  "(" Type() ")"
UnaryExpressionNotPlusMinus()
}

void PrimaryExpression() :
{}
```

```
{
  PrimaryPrefix() ( LOOKAHEAD(2)
PrimarySuffix() )*
}

void PrimaryPrefix() :
{}
{
  Literal()
|
  "this"
|
  "super" "." <IDENTIFIER>
|
  "(" Expression() ")"
|
  AllocationExpression()
|
  LOOKAHEAD( ResultType() "." "class" )
  ResultType() "." "class"
|
  Name()
}

void PrimarySuffix() :
{}
{
  LOOKAHEAD(2)
  "." "this"
|
  LOOKAHEAD(2)
  "." AllocationExpression()
|
  "[" Expression() "]"
|
  "." <IDENTIFIER>
|
  Arguments()
}

void Literal() :
{}
{
  <INTEGER_LITERAL>
|
  <FLOATING_POINT_LITERAL>
|
  <CHARACTER_LITERAL>
|
  <STRING_LITERAL>
|
  BooleanLiteral()
|
  NullLiteral()
}

void BooleanLiteral() :
{}
{
  "true"
|
  "false"
}

void NullLiteral() :
{}
{
  "null"
}

void Arguments() :
{}
{
  "(" [ ArgumentList() ] ")"
}

void ArgumentList() :
{}
{
  Expression() ( "," Expression() )*
}

void AllocationExpression() :
{}
{
  LOOKAHEAD(2)
  "new" PrimitiveType() ArrayDimsAndInits()
|
  "new" Name()
    (
      ArrayDimsAndInits()
    |
      Arguments() [ ClassBody() ]
    )
}

/*
 * The second LOOKAHEAD specification below
is to parse to PrimarySuffix
 * if there is an expression between the
"[...]".
 */
void ArrayDimsAndInits() :
{}
{
  LOOKAHEAD(2)
  ( LOOKAHEAD(2) "[" Expression() "]" )+ (
LOOKAHEAD(2) "[" "]" )*
|
  ( "[" "]" )+ ArrayInitializer()
}


/*
 * Statement syntax follows.
 */

void Statement() :
{}
{
  LOOKAHEAD(2)
  LabeledStatement()
|
  Block()
|
  EmptyStatement()
|
  StatementExpression() ";"
|
  SwitchStatement()
|
  IfStatement()
|
  WhileStatement()
|
  DoStatement()
|
  ForStatement()
|
  BreakStatement()
|
  ContinueStatement()
```

```
|
  ReturnStatement()
|
  ThrowStatement()
|
  SynchronizedStatement()
|
  TryStatement()
}

void LabeledStatement() :
{}
{
  <IDENTIFIER> ":" Statement()
}

void Block() :
{}
{
  "{" ( BlockStatement() )* "}"
}

void BlockStatement() :
{}
{
  LOOKAHEAD([ "final" | Type()
<IDENTIFIER>)
  LocalVariableDeclaration() ";"
|
  Statement()
|
  UnmodifiedClassDeclaration()
}

void LocalVariableDeclaration() :
{}
{
  [ "final" ] Type() VariableDeclarator() (
"," VariableDeclarator() )*
}

void EmptyStatement() :
{}
{
  ";"
}

void StatementExpression() :
/*
 * The last expansion of this production
accepts more than the legal
 * Java expansions for StatementExpression.
This expansion does not
 * use PostfixExpression for performance
reasons.
 */
{}
{
  PreIncrementExpression()
|
  PreDecrementExpression()
|
  PrimaryExpression()
  [
    "++"
  |
    "--"
  |
    AssignmentOperator() Expression()
  ]
}

void SwitchStatement() :
{}
{
  "switch" "(" Expression() ")" "{"
    ( SwitchLabel() ( BlockStatement() )*
)*
  "}"
}

void SwitchLabel() :
{}
{
  "case" Expression() ":"
|
  "default" ":"
}

void IfStatement() :
/*
 * The disambiguating algorithm of JavaCC
automatically binds dangling
 * else's to the innermost if statement.
The LOOKAHEAD specification
 * is to tell JavaCC that we know what we
are doing.
 */
{}
{
  "if" "(" Expression() ")" Statement() [
LOOKAHEAD(1) "else" Statement() ]
}

void WhileStatement() :
{}
{
  "while" "(" Expression() ")" Statement()
}

void DoStatement() :
{}
{
  "do" Statement() "while" "(" Expression()
")" ";"
}

void ForStatement() :
{}
{
  "for" "(" [ ForInit() ] ";" [
Expression() ] ";" [ ForUpdate() ] ")"
Statement()
}

void ForInit() :
{}
{
  LOOKAHEAD( [ "final" | Type()
<IDENTIFIER> )
  LocalVariableDeclaration()
|
  StatementExpressionList()
}

void StatementExpressionList() :
{}
{
  StatementExpression() ( ","
StatementExpression() )*
}
```

```
void ForUpdate() :
{}
{
  StatementExpressionList()
}

void BreakStatement() :
{}
{
  "break" [ <IDENTIFIER> ] ";"
}

void ContinueStatement() :
{}
{
  "continue" [ <IDENTIFIER> ] ";"
}

void ReturnStatement() :
{}
{
  "return" [ Expression() ] ";"
}

void ThrowStatement() :
{}
{
  "throw" Expression() ";"
}

void SynchronizedStatement() :
{}
{
  "synchronized" "(" Expression() ")"
Block()
}

void TryStatement() :
/*
 * Semantic check required here to make
sure that at least one
 * finally/catch is present.
 */
{}
{
  "try" Block()
  ( "catch" "(" FormalParameter() ")"
Block() )*
  [ "finally" Block() ]
}
```

VITA AUCTORIS

Xiaohong Yang was born in 1968 in Wuhan, China. She graduated from Huazhong University of Science & Technology, Wuhan, P.R.China in 1990, where she received a Bachelor's degree in Computer Engineering. From there she went to work in an electronic technology Company.

In 1992, she went back to Huazhong University of Science &Technology, and received a Master's degree of Computer Engineering in 1995. She is currently a candidate for the Master's degree in Computer Science at the University of Windsor, and hopes to graduate in the summer of 2001.