Object-Oriented Program Testing using Formal Requirements Specification

by

Senthil K. Subramanian

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Master of Science
in
Computer Science

Winnipeg, Manitoba, Canada, 1997

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

OBJECT-ORIENTED PROGRAM TESTING

USING FORMAL REQUIREMENTS SPECIFICATION

BY

SENTHIL K. SUBRAMANIAN

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

I hereby declare that I am the sole author of this thesis.

I authorize the University of Manitoba to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Manitoba to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Abstract

Research on object-oriented paradigm has been mainly focussed on developing object-oriented programming languages and object-oriented analysis design tools. Recently, other aspects in object-oriented software life cycle have received attention because of the perceived importance provided to object-oriented software development. Testing is one of the phases which warrants attention because software quality partly depends on testing.

Testing methods can be broadly classified into two categories: specification-based and program-based testing. In specification-based approach, test cases are derived from the requirements specification of a software system while in the other approach, they are derived from the semantics of the programming language chosen for implementation. As requirements analysis precedes design in a life cycle model, specification-based test cases will be independent of the programming language and hence can be used on several designs and/or implementations of the same application. The introduction of formal methods, particularly in the requirements analysis of the software life cycle, has facilitated more research in specification-based testing techniques. In this thesis, we present a method to generate test cases for an object-oriented software from its requirements specification. The Object-Z formal specification language has been used to specify the requirements and Object Modeling Technique (OMT) proposed by Rumbaugh and others has been used as the design notation. The method was successfully applied to two different case studies: a

Library Management System and an Automated Teller Machine, both of which were engineered through the software development life cycle. The generated test cases were used to validate two different implementations, one in C++ and the other in Smalltalk. We also address the issue of polymorphism and the resulting complexity to testing. In the proposed method, the presence of an application domain model at both the specification and design level, provides a clear understanding of the system under consideration. The advantages of the proposed method have been elaborately discussed in the thesis.

Acknowledgements

There are a number of people whom I would like to thank for enabling me in making this thesis a reality. I am grateful to my advisor, Dr. Kasi Periyasamy for his relentless and untiring support in all possible ways while doing this thesis. Also I would like to express my thanks to my colleagues in the Software Engineering group for their useful tips during the research meetings. I would also like to thank my friends in the department and outside, who provided moral support.

Contents

List of Figures

Chapter 1

Introduction

The verification and validation phase in the software development life cycle is important because its goal is to convince the end users that the software satisfies its requirements. As the need for the production of high quality software increases, more importance is being given to the verification and validation phase; a high quality software will effectively reduce maintenance cost associated with the software. The objective of the verification and validation phase is therefore to detect errors in the code and ensure reliability in the software being developed. *Verification* refers to the set of activities that ensure that a software correctly implements a specific function. *Validation* refers to the process of convincing end users that the software meets its requirements. Validation is also called testing; consequently, a validation process refers to exercising a set of test cases on the implementation to ensure that there are no errors. This thesis is a contribution to object-oriented software testing.

Testing process involves controlled exercise of a program using data in

order to expose the errors. The intent of a testing technique is to ensure that software works correctly. By isolating errors, testing provides a valuable measure of the quality of software. A testing process ideally requires several test cases and it is a laborious process due to the number of test cases that need to be generated. It is also expensive and time-consuming when compared to other phases in the life cycle. This is because of the number of test cases that must be considered. Moreover, the formulation of a testing framework that is required to generate test cases involves arduous effort. Despite these shortcomings, testing plays a crucial role in the development process because the quality of software depends to a major extent on testing.

Typically, a testing process consists of four steps: i) identifying the criteria for testing; ii) generating test cases which are comprised of input and expected output; iii) executing the test cases (i.e., applying them); and iv) comparing the results obtained, with the expected output and tabulating them.

1.1 Types of Testing Methodologies

Testing is normally carried out after the implementation is complete. The existence of defects or inadequacies to capture the requirements in a program can be inferred from its output. In order to completely validate a program, theoretically, all possible inputs must be exercised on it. Because of this impractical scenario, there is a need for systematically generating test cases which must generate as few test cases as possible . Testing methods can be broadly classified into two categories: *specification-based* and *program-based* testing.

Specification-based Testing (Black box testing)

The growing interest in the use of formal specifications in software development has raised interest in probing its resourcefulness in testing. In this approach, test cases are derived from the requirements specification of the software. A test case typically consists of i) the entity to be tested (e.g., variable, statements, expression etc.,) , ii) test data that must be exercised, and iii) the expected output. As test cases are generated from the requirements, they will be independent of the programming logic and algorithms that are chosen for implementation. Hence, they can be used for all implementations of the same application. In summary, specification-based testing concentrates on application-domain information. Testing strategies such as *Cause-effect testing* and *domain testing* belong to the category of specification-based testing [22].

Program-based Testing (White box testing)

This approach of software testing is intended to test the internal structure of a program. Test cases generated from programs are based on the semantics of the programming language used for implementation. As concentration is on the code, this testing technique can be used to uncover errors in the logic of programming. Hence, it will ensure confidence in the implementation. As an example, the structure of a program can be used to derive test cases. Depending on the quantification of quality assurance mechanisms, program-based testing can be applied to either whole or portions of an implementation. For non-trivial segments of code, testing must be carried selectively depending upon the require-

ments which may require tools such as dynamic program analyser. Traditional program-based testing techniques include *path testing*, *data flow testing*, *state-based testing*, *McCabe cyclomatic testing*, *mutation testing*.

Previous research in software testing concentrated on program-based testing. Traditionally, requirements of a software component were specified using diagrammatic techniques such as data flow diagrams which do not provide adequate support for generating test cases. Recently, with the introduction of formal methods in software engineering, particularly in the requirements analysis and design phases of a life cycle model, more research is being done on specification-based testing techniques. In addition, the advent of tool support for formal methods make test case generation using specification-based testing techniques relatively easier and justifiable. But for a total coverage of testing, both specification-based and program-based testing are required.

1.2 Object-oriented Approach

Object-orientation is becoming popular due to its advantages in maintenance. iterative development and reuse. Compared to the functional paradigm, where emphasis is more on functions or procedures, object-oriented paradigm emphasizes on data types and operations encapsulated within each data type. The separation of data types and operations that manipulate them supports information hiding; this is helpful in modifying the implementation of internal functions independently. In general, modeling a system as a collection of ob-

jects associated with their identity and behavior facilitates the construction of highly modular systems.

The fundamental difference between functional and object-oriented paradigms therefore lies in decomposing the problem domain. A task-oriented view is followed by the funtional approach while a data-centered modeling point is adopted in the latter. Another major difference is that in the functional paradigm, a design representation is generally different from that used in the analysis. This clearly represents a shift in focus and also demarcates analysis from design phase in the software life cycle. But in object-oriented paradigm, software development proceeds through a seamless transition of the models which are formulated in the analysis phase and are carried over to other phases in the development life cycle.

In summary, the advantages of object-oriented approach can be briefly enumerated as below:

- Object-oriented software is modeled around objects and their interactions. This leads to better understanding of the problem domain due to separation of concerns.
- Almost all object-oriented life cycle models provide uniform structure throughout the life cycle. This structure is abstract in the requirements analysis phase and is refined into concrete entities in the design and implementation phases.
- During maintenance phase, tracing objects and localizing the effect of an operation becomes relatively easier. Moreover, expansion of the sys-

tem can be made consistently with less effort, due to inheritance and polymorphism.

- Support for code reusability is offered by almost all object-oriented models.

1.3 Testing object-oriented software

Previous research on object-oriented software development focussed mainly on developing object-oriented programming languages and later on developing tool support for analysis and design. The testing phase of object-oriented software has not received much attention. Though, an object-oriented approach leads to the construction of better quality programs by efficient analysis and design techniques, testing is still essentially required in ensuring high quality. Further as the concept of *reusability* is strongly advocated, a more rigorous approach for testing object-oriented software becomes mandatory. The methodologies and tools that were used to test software developed using the functional paradigm are largely inadequate to test object-oriented software, mainly due to the distinguishing characteristics of object-orientation, such as *encapsulation* and *inheritance*.

As stated earlier, object-orientation focuses on organizing a system as a collection of objects. An object encompasses its state and a set of operations that define its behavior. The operations act on an object's state and possibly modify them. Testing an object therefore requires testing its state and behavior. In addition, interactions among objects which also contribute to the

functionality of the system must also be tested.

The distinguishing features of object-oriented paradigm such as *encapsulation*, *inheritance* and *polymorphism* increase the complexity of testing [12]. The problems associated with testing object-oriented software have been discussed in [1]. The compartmentalization of a system in terms of objects is facilitated by encapsulation. However, the resulting opacity created as a result of information hiding mechanism will affect testing. When inheritance is used, the inherited features can be modified and new properties can be added by the inherited class. This creates anan additional problem as to whether the inherited features need to retested in the context of the derived class. Polymorphism brings in the concept of undecidability as to which code will be selected during execution.

Due to these salient features of object-orientation, testing techniques that were used to test programs developed using functional paradigm cannot be used directly to test object-oriented software. Hence the need to develop additional methods and/or methodologies to test object-oriented software became mandatory.

1.4 Related Work

Research on object-oriented software testing initially focussed on *program-based testing.* Program-based testing techniques for object-oriented programs were developed based on the same principles as those used for testing functional programs. Class-based testing as proposed by Hoffman and others [10, 11]

belongs to the category of program-based testing. A class has type information which covers all values that objects of this class can possess. Therefore, testing a class will exhaustively cover all objects of that class. Test graphs are used to identify relationships that exist between classes. McGregor and others [9] have proposed a methodology which is based on testing functional programs. This takes into consideration the hierarchical nature of class evolution to reuse test cases admirably. Turner and his colleagues [24] concentrated on dynamic behavior of objects and proposed the use of state-based testing technique.

Research on specification-based testing is relatively new; they seem to be centered around notations of the specification languages. For example, the work reported in [8] is based on the specification language Estelle and is used for telecommunication purpose. Recently, Barbey and others used algebraic specification technique to test Ada data types [2]. But this research is still in its primitive stage. Use of model-based specification techniques such as Z for testing has been reported in [3], which describes the theory behind specification-based testing as well. In [22], two new specification-based testing strategies - *domain propagation* and *specification-mutation* have been explained. Domain propagation is based on an extension to the partition testing technique, which is used to partition the input space into domains according to some criteria. Input domains are partitioned which map to the identified output domains while the output partition is determined by reducing the output expression to a disjunctive normal form. In the latter technique, mutant programs are introduced in test cases and testing is achieved by determining how many mutants are distinguished. The success of this technique largely depends on the tests

which are passed earlier. Use of a model based specification-language clearly aids in generating test cases for establishing the validity of operations involved in a system. The use of specifications to create test suites has lead to more research on *reusability* of test cases.

Most of the object-oriented specification testing techniques do not provide any guidelines on their usage [22]. Further, in order to confirm the functionality of a whole system, test cases must be generated using a methodical approach so that sequencing of operations can also be tested. Object-oriented formal languages are quite new in application and hence there is not much research done on testing based on object-oriented specification techniques.

1.5 Focus of this thesis

In this thesis, we propose a more rigorous and systematic testing method for generating test cases from object-oriented formal requirements specification. The specification language Object-Z, which is an extension to Z, has been used in this thesis.

The methodology consists of the following steps :

- Specify the requirements of the software in Object-Z.
- Derive an object-oriented design from the requirements specification. Object Modeling Technique proposed by Rumbaugh [19] and others has been used to model the design.
- Derive test cases from the requirements specification, and the design.

- Implement the object-oriented design using an object-oriented programming language.

- Exercise the test cases on the implementation.

This thesis also addresses the issue of *polymorphic substitutions* that occur in object-oriented programs. Polymorphism brings in undecidability because of dynamic binding that takes place at run-time. In object-oriented programs, *polymorphism* can occur in two ways: (i) A method's name may be bound to more than one code; for example, within a class C, a method *abs* may be defined twice, one for real numbers and another for complex numbers. (ii) A variable name v may be bound to objects of different classes, one at a time. This situation occurs when v is declared to be an object of a superclass and at run-time it is bound to an object of a subclass. In both cases, testing a polymorphic function or polymorphic variable requires the testing of all possible substitutions as well.

The major problem in testing polymorphic substitutions is therefore to reduce the number of test cases that must be generated. In case of polymorphic functions, the common code for all versions of the same function can be identified which will lead to some simplification. However, this is not guaranteed in all situations. It depends on the application and the way code has been written. The second case of polymorphism occurs due to inheritance relationship. Since a subclass inherits all features of the superclass, it is possible to reuse the test cases developed for methods in the superclass for testing the methods in the subclass. However, this approach is applicable only for - *subtyping or*

inheritance by expansion. The use of formal specification not only enables us to derive application-oriented test cases, but also provides a formal basis to reuse test cases for subclasses to test superclasses. This is because a subclass object must behave identically with respect to its superclass object. If the subclass is permitted to redefine a method, then the common characteristics between the redefined method and the original method can be identified and the proposed method can be partially used. This is not addressed in detail in this thesis and is reserved as a future work.

Recently there has been some work done in minimizing test cases under polymorphic substitutions. McDaniel [15] has proposed the use of an existing testing technique called Robust Testing, to simplify the number of test cases which are necessary for testing polymorphic substitutions. The suggested approach makes use of an orthogonal array in defining the variables that are involved in a polymorphic interaction. Amit [16] has suggested the use of an Augmented Object Relationship Diagram (AORD) to test polymorphic interactions. This approach resembles the one using call-graph for identifying the interactions. The entities in an AORD contain information about attributes and/or methods of their respective classes.

The organization of this thesis is as follows: Chapter 2 describes briefly about the Object-Z notation. The Object Modeling Technique is discussed in Chapter 3. Chapter 4 describes the proposed methodology in detail. A formalized notation of the testing methodology is also provided in this chapter. In the last chapter, salient features of the test case generation technique is discussed. Also a comparison is made with other existing testing techniques.

Details regarding two case studies, a Library Management System and an Automated Teller Machine, the generation of test cases for both case studies have been enclosed in the Appendices.

Chapter 2

The Object-Z Formal Notation

The prevailing methods of documenting software requirements are not precise and are generally ambiguous. *Formalization* of the requirements is quite helpful in comprehending the problem by providing a precise and unambiguous description. A formal specification of requirements describes the properties of the system to be designed without constraining the way those properties must be achieved.

Formal methods use mathematical notations to describe the behavior of the system being modeled. These notations have precise syntax and well defined semantics. Being independent of the code, a formal specification offers a great deal of flexibility in the analysis and design phases. Implementation- oriented details need not be brought in too early in the development process. In addition, the requirements specification can also be used during maintenance.

In this thesis, the object-oriented formal specification language Object-Z[18] has been used. Object-Z is an object-oriented extension of the Z specification language [21].

2.1 Z Notation

The Z notation is a model-based specification language and uses abstract mathematical data types to model data in a system. It is based on *typed set theory* and hence every value defined in the specification must belong to a certain type. The types are carrier set and specify the range of values that a particular variable can possess. There are three primitive types in Z - natural number ($\mathbb{N}$), natural number excluding zero ($\mathbb{N}_1$) and integer ($\mathbb{Z}$). Apart from these primitive types, users can also define their own types which are called basic types. The basic types will be refined in latter stages of the design process and/or during the course of implementation.

Z supports two types of abstractions: *representational abstraction* and *behavioral abstraction*. The former describes static declarations in a specification such as global constants, types and state spaces and the latter describes functions and operations of the system.

A Z specification facilitates splitting of a whole system into manageable pieces called *schemas*. These schemas describe either static or dynamic aspects of the system. The static aspects define the state space and relationships among the variables in the state space. Any state changes due to possible operations on the state and input/output relationships are captured by the dynamic model. A schema is identified by a unique name. It comprises a declaration (signature) and optionally a predicate part (properties). The declaration part consists of local variables and their types, denoting the structure of the schema; predicate part expresses static relationships among the

variables in the declaration part. The variables in the predicate part can also be global variables apart from the locally defined variables that are within the scope of the current schema. Requirements that are stated in the predicate part are generally called as *invariants* and must be true for any state changes that are effected.

To illustrate, consider the information concerning a student in a university environment. This is defined by the following schema.

Student

$name : String$
$registration\_no : \mathbb{N}_1$
$year\_of\_joining : Date$

$year\_of\_joining >= 1992$

The above schema contains three variables: *name*, *registration_number* and *year_of_ joining* and a predicate asserting that the student must have joined in 1992 or after.

An operation in Z is also defined using schema notation. It indicates the state space over which the operation acts on. In the declaration part of an operation schema, names of the state components before and after the operations are specified through unprimed and primed variables respectively . Input and output variables for the operation are indicated with a trailing question mark (?) and an exclamation mark(!) respectively. The predicate part of the operation schema asserts pre- and post-conditions of the operation. As an example, the operation to add a student is given below:

AddStudent

$$
\begin{array}{|l}
\hline
student, student' : \mathbb{P}\, Student \\
newstudent? : Student \\
\hline
\neg\, (\exists\, stud : Student \bullet stud \in student \land \\
\qquad stud.registration\_no = newstudent?.registration\_no) \\
student' = student \cup \{newstudent?\} \\
\hline
\end{array}
$$

The declaration part of the above operation schema indicates two state components *student* and *student'*; these two indicate respectively the state before and after the operation. In addition, *AddStudent* also introduces an input variable *newstudent?*. The pre-condition asserts that there is no student in the system with the same registration number and the post-condition asserts that the new student is added to the system. Other intrinsic details of operation schemas can be found in [17, 21].

For specifying complex operations, the Z notation allows the manipulation of schema operations using logical operators. This facilitates the system to be developed using either a top-down or a bottom-up approach.

2.1.1 Functions

Functions, relations and global constants are all defined using the notation for axiomatic definition. A function is a special case of a relation in which no element in the domain is mapped to more than one element in the range. It is similar to a function in procedural languages such as C or Pascal. A function has a signature and definition. The signature consists of the names and

types of both input and output parameters and the definition part expresses some relationship between the domain and range of the function. For example, given an instance of *Student* schema, a projection function to return the *year_of_joining* can be specified as follows:

$$Date\_of\_Joining : Student \rightarrow Date$$
$$\forall st : Student \bullet Date\_of\_Joining(st) = st.year\_of\_joining$$

2.2 Object Z Notation

Object-Z is an object-oriented extension to Z. A typical Z specification consists of a number of states and operation schemas. Inferring which operations affect which state space is feasible on examining the signatures of the operation. In contrast to this, an Object-Z specification of a system consists of a set of global definitions and a set of class definitions. The global definitions are shared by all the class definitions. A class definition is encapsulated; i.e, the entities that are present within a class definition are local to the class itself. Global definitions and the paragraphs within each class definition are written using the Z notation. The general structure of a class definition in Object-Z is given in Figure 2.1.

The *visibility list* of a class C lists all the features that are exported from C. If omitted from the definition, all features of C are visible. Typically, this list includes attributes and operations of C. The *list of inherited class names* of

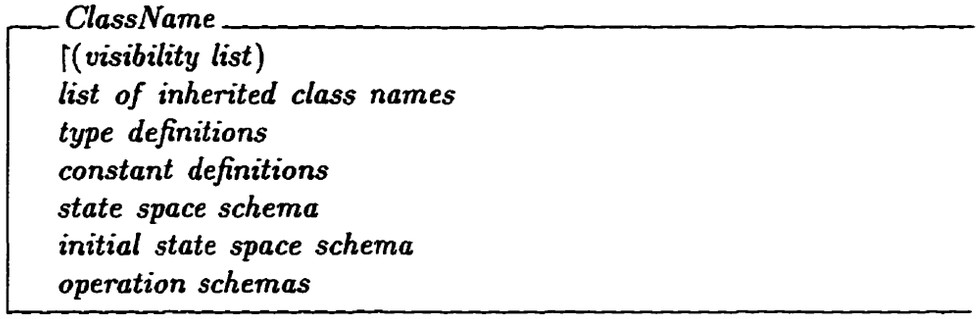


Figure 2.1: Structure of a class definition in Object-Z

C indicates all the superclasses of C. While inheriting a superclass C_{sup} into C, the *state space schema*, *initial state space schema* and *operation schemas* of C_{sup} are conjoined with the respective components in C according to schema conjunction in Z. Redefinition of an operation can only provide additional constraints and/or introduce new parameters. The *visibility list* of C_{sup} is not inherited into C. Hence C can export an inherited feature from C_{sup}, although C_{sup} does not export that feature. *Type definitions* and *constant definitions* are similar to those in Z, but are local to the class. The *state space schema* is unnamed; the declaration part of the *state space schema* corresponds to attributes of the class. The *initial state space schema* describes the set of all initial states of objects instantiated from the class.

Operation schemas are defined as in Z, with one exception. While an operation in Z includes the state spaces which are affected by the operation using the Δ or Ξ notation, the state space is implicit for an operation in Object-Z; the state space of the class. The notation $\Delta(x, y)$ in an operation

Op in Object-Z asserts that the attributes x and y are affected by *Op*; other attributes of the class are unaffected by *Op*. This is a major semantic difference between Z and Object-Z.

Operations in Object-Z can also be defined by composing several other operations. There are four composition operators in Object-Z. These are: *conjunction* ($\wedge$), *parallel* ($\|$), *sequential composition* ($\fatsemi$) and *choice* ($[]$). The expression $Op_1 \wedge Op_2$ denotes an operation schema in which the operations Op_1 and Op_2 are independent and will be invoked concurrently; common declarations in both operations are equated. The expression $Op_1 \| Op_2$ denotes hand-shaking communication between the operations Op_1 and Op_2. So, input variables of one operation are equated to the output variables of the other as long as they have the same base name (e.g., $x?$ and $x!$). The expression $Op_1 \fatsemi Op_2$ denotes sequential composition in which Op_2 follows Op_1. Accordingly, Op_2 proceeds only after Op_1 terminates; the notion of intermediate state is evident. The expression $Op_1 [] Op_2$ denotes an operation in which either Op_1 or Op_2, but not both, will be enabled depending on the pre-condition of whichever operation is satisfied. If both the pre-conditions are satisfied, one of the operations is non-deterministically chosen. Object-Z notation is case-sensitive. Hence, the identifiers *student* and *STUDENT* are different; the former may denote an object of a class while the latter may denote a value of an enumerated constant. An example specification for a system has been provided in the appendix.

In Object-Z new classes may be derived from one or more existing classes through inheritance by expansion. In the derived class, the state schemas and

the initialization schemas are conjoined with those in the derived class. Operations which have the same name in the derived class is also treated in the same manner. Any unavoidable name clashes must be resolved by renaming in the derived class. This type of inheritance leads to polymorphism. Through polymorphism, a variable of the base class type can be substituted by variables of derivative class types without causing any difference in the intended functionality of the operation. This leads to undecidability and ultimately increases the semantic complexity of the the system. A more detailed description of Object-Z can be found in [18].

Chapter 3

Object Modeling Technique (OMT)

The Object Modeling Technique (OMT) described by Rumbaugh and others [4, 19] is used as the design notation in this thesis. This methodology with a series of well-defined steps helps in object-oriented software development, and can be used throughout the software life cycle. Essential aspects from the application domain are extracted and are modeled in the analysis phase. The design phase concentrates on concretizing the modeled entities. OMT supports the modeling of an application in three different perspectives: *object*, *dynamic* and *functional*. Each model evolves during the software development process and hence all of them are necessary during different stages in the life cycle.

Object Model

An object model describes the static structure of the system being modeled. This model consists of a collection of objects and their relationships with other objects in the system. It captures attributes as well as operations of each object. Since objects are run-time instantiations of classes, one would typically describe classes and static relationships among classes in an object model. However, OMT does permit a separate notation to represent objects, though it is rarely used. The object model is essentially driven by relevance to the application domain and provides an essential frame work based on which dynamic and functional models are developed. Hence, formulation of an object model precedes the other two models. As an example for object model, consider a simple model of a *registration system* in a university-environment as shown in Figure 3.1.

The model captures four classes: *student*, *courses registration system* and *student council president*. There is an inheritance relationship between *student* and *student council president*. The registration system aggregates *student* and *course*. Figure 3.1 also shows multiple associations between *student* and *course* classes.

Dynamic Model

In the dynamic model, time variant activities of the system are captured. This model consists of a collection of state diagrams. A state diagram characterizes states and state transitions of an object in response to some event. Thus, it describes the behavior of objects instantiated from a

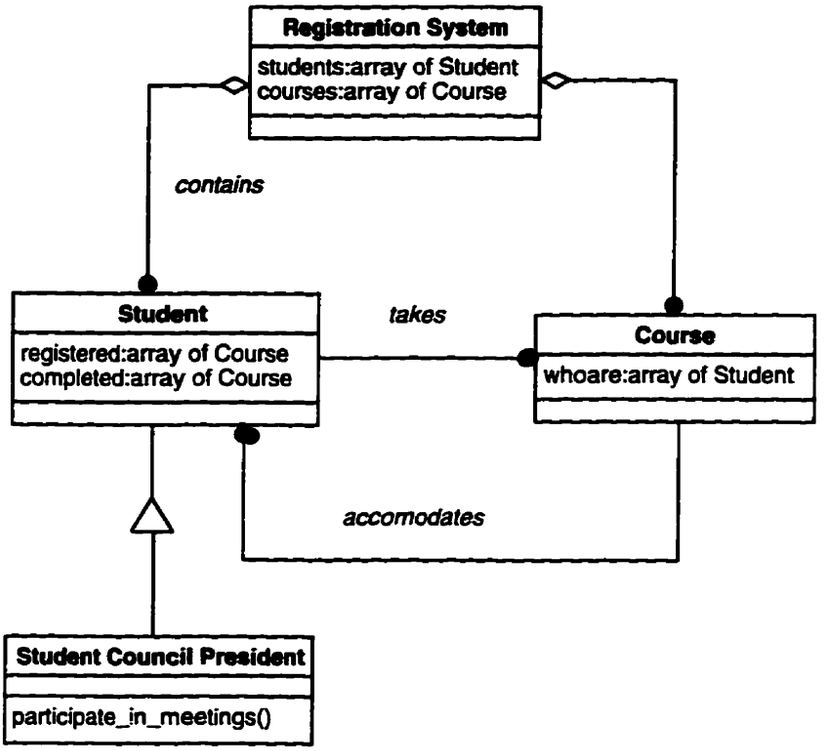


Figure 3.1: Student class in a Registration System

single class. In short, the dynamic model effectively achieves abstraction of the pattern of events, states and state transitions for a particular class in a system. The notion of generalization as supported by inheritance mechanism is also available in the dynamic model. That is, transitions of a superstate can be inherited by several substates.

A transition shows the effect of exercising an operation. It may cause a state change or may cause changes within a state which are not considered to be significant. In the former case, the transition links a source state and a target state; the object will be in the target state after the

transition. Generally, a transition is fired by an event. A transition may be associated with a condition in which case the transition is fired only if the condition is satisfied. It may also contain action and activities *.

To illustrate, consider the state transition diagram of *student* class given in the object model earlier, which has been depicted in Figure 3.2. It has three significant states; *Joining* refers to the state when the student has just joined and has not registered for any course and hence has not completed any course either. *InProgram* depicts the state of student when the student has completed some courses and also has registered for some more courses; *Graduating* refers to the state when the student has completed all courses necessary for a program and hence is able to graduate. Notice that in *Joining*, the transition *register* causes a state change, whereas while in *InProgram*, the same transition does not lead to any state change. The state *Graduating* is associated with an activity called *graduate*.

Functional Model

In the functional model, functional dependencies between input and output values for operations in the system are modeled. This is achieved by using Data Flow Diagrams (DFDs). Typically, this model characterizes the transformation of data. These transformations correspond to *actions* and *activities* in the dynamic model and *operations* on objects in the object model. This model represents macro-level functionality [13]. A functional model can also be nested hierarchically. Unlike the

*OMT distinguishes between an action and an activity; the former consumes negligible time for execution whereas the execution duration of an activity is observable

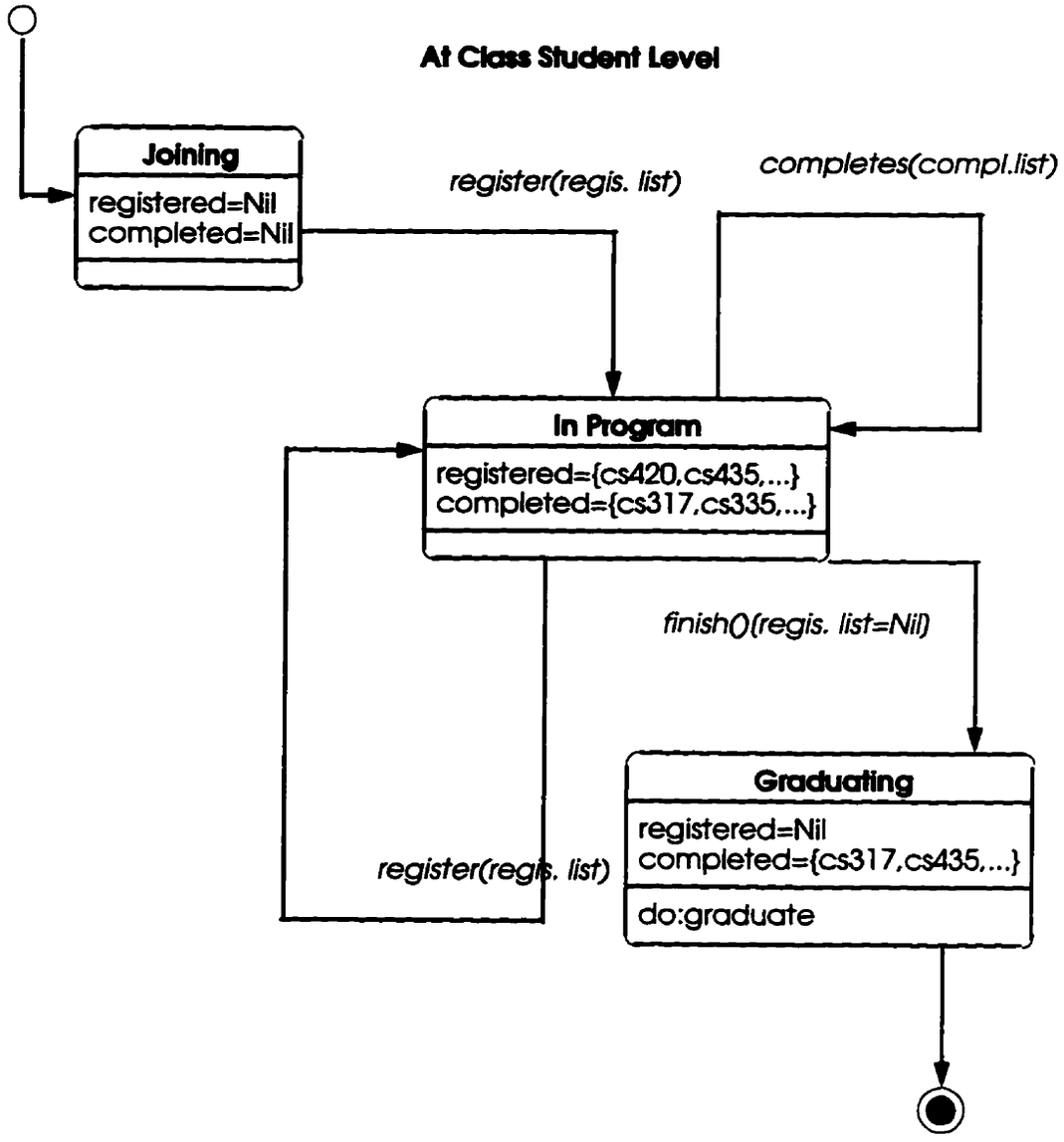


Figure 3.2: State Transition Diagram for Student

dynamic model, there is one simple logical diagram that depicts the functional model, even though it can span over several physical pages. See

the functional model for the registration system shown in Figure 3.3.

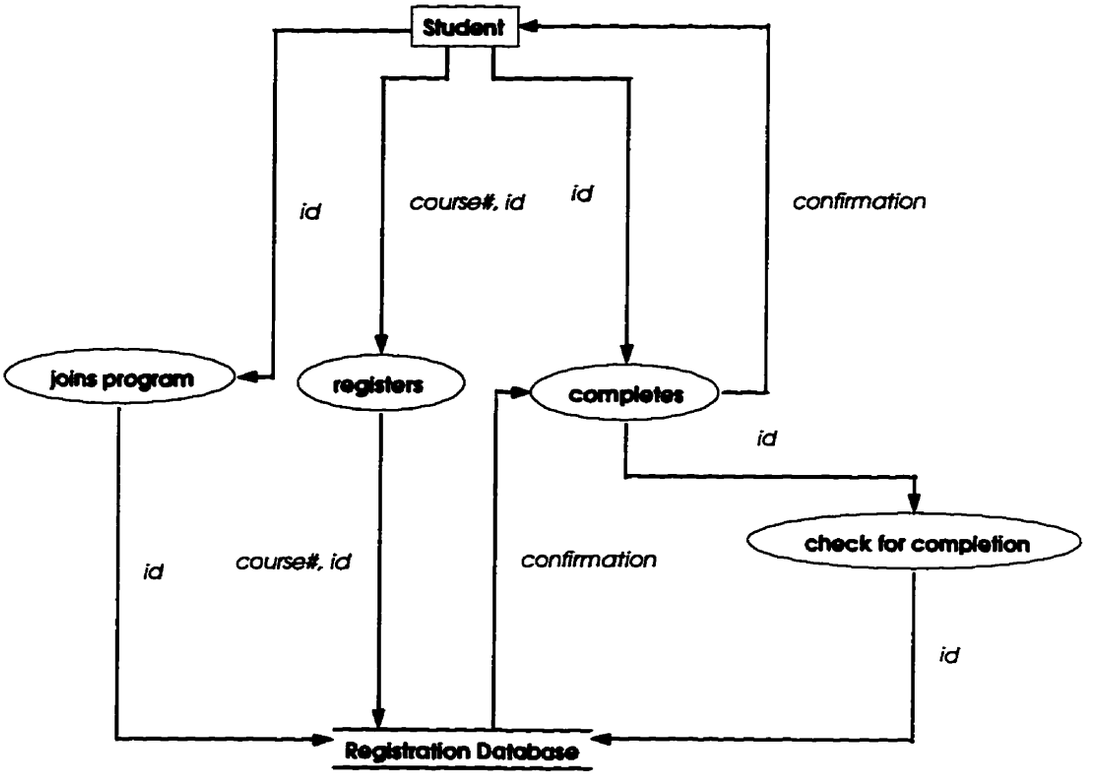


Figure 3.3: Functional Model of Registration System

3.1 Deriving Design from Object-Z Specification

In this section, we describe about deriving an object-oriented design from an Object-Z specification. As stated earlier, the design will be represented using OMT. Even though both specification and design notations are object-oriented, there still exists some subtle changes during the derivation process. These changes are regarded as design decisions which also influence test case generation later.

3.1.1 Deriving Object Model

A class in Object-Z is directly mapped to a class in OMT with its state variables being transformed as attributes. The operations are also transformed in a similar manner. Functions within a class definition are local to the class; these can be transformed as internal operations in the design.

Type definitions in Object-Z can be mapped in two ways: i) a separate class can be created in the design with a local structure and all relevant functions from the specification can be transformed into the new class; ii) additional structure can be created within the class corresponding to the class in the specification. Depending on the need for ease of understanding, we follow either of the two approaches.

Figure 3.4 shows an example for transforming a class definition in Object-Z to a class in the design. In this case, we keep the free type definition outside

the class during transformation.

The inheritance relationship in specification can be represented directly using the symbol for generalization in OMT. Since the specification does not indicate aggregation directly, it is the responsibility of the designer to extract this information.

For association, we define the transformation as follows: if a class C has an operation Op which contains a parameter or a local variable of type B, B being another class, we establish an association relationship between classes, C and B. This is justified from the definition of association in OMT [19] (i.e., semantic dependency between the classes) and the using relationship in Booch's [4] approach. The designer should add multiplicity of the association, role of the association and qualifiers, if these can be derived from the application domain. Such information will not be generally available in Object-Z specification.

3.1.2 Deriving Dynamic Model

A state transition diagram generally contains two or more states. Since diagrams with less than two states do not convey any significant information, they are ignored from discussion. In addition, a specification does not directly describe the state of an object, and hence it is the designer's responsibility to extract information regarding states from the specification. There are two clues provided by an Object-Z specification to identify the states: *an initial state*, if present in the specification, will lead to an initial state in the design. This initial state in the design will contain appropriate actions and activities

to initialize the attributes of the class, as indicated by the initial state in the specification. The second clue is the Δ notation, present in an operation. When an operation changes the state variables, there occurs a state change. This may correspond to a transition leading to a different state or back to the same state again. The significance of the change in the variables must therefore be interpreted by the designer based on the application domain.

Transitions in a state transition diagram can be directly inferred from the operations in the specification. As mentioned in the previous section, functions present in a specification are also transformed into operations (generally as internal operations) but not all of these operations lead to transitions. Some may become actions; others may become activities. We do not discuss more about this justification in this thesis.

The state invariant and the predicate part of operations in the specification play a vital role in the transformation. Most of the conditions that are associated with states and transitions are derived from these assertions. In particular, the state invariant asserts conditions that must be true at every state of the object. The pre-condition of an operation provides the conditions for the transition(s) which is(are) derived from the operation. The post-condition gives the clue to derive activities of the state to which the transition leads, and also the events generated as a result of performing the transition.

It must also be noticed that an operation in the specification serves as a source for the transitions, conditions and events. Consequently, there need not be a one-to-one correspondence between the operation and a transition. Since the focus of the thesis is on deriving test cases, rather than justifying

the design derivation process, we do not elaborate the derivation process any further in this thesis.

Our approach does not include the functional model of OMT. The two major reasons for ignoring the functional model are: i) it simply provides another perspective, particularly functional perspective, of the application; ii) none of the other object-oriented design methods, including the newly emerging Unified Modeling Language (UML) [5, 6] supports a functional model. We believe that both the object and dynamic models of OMT sufficiently cover all required information for test case generation.

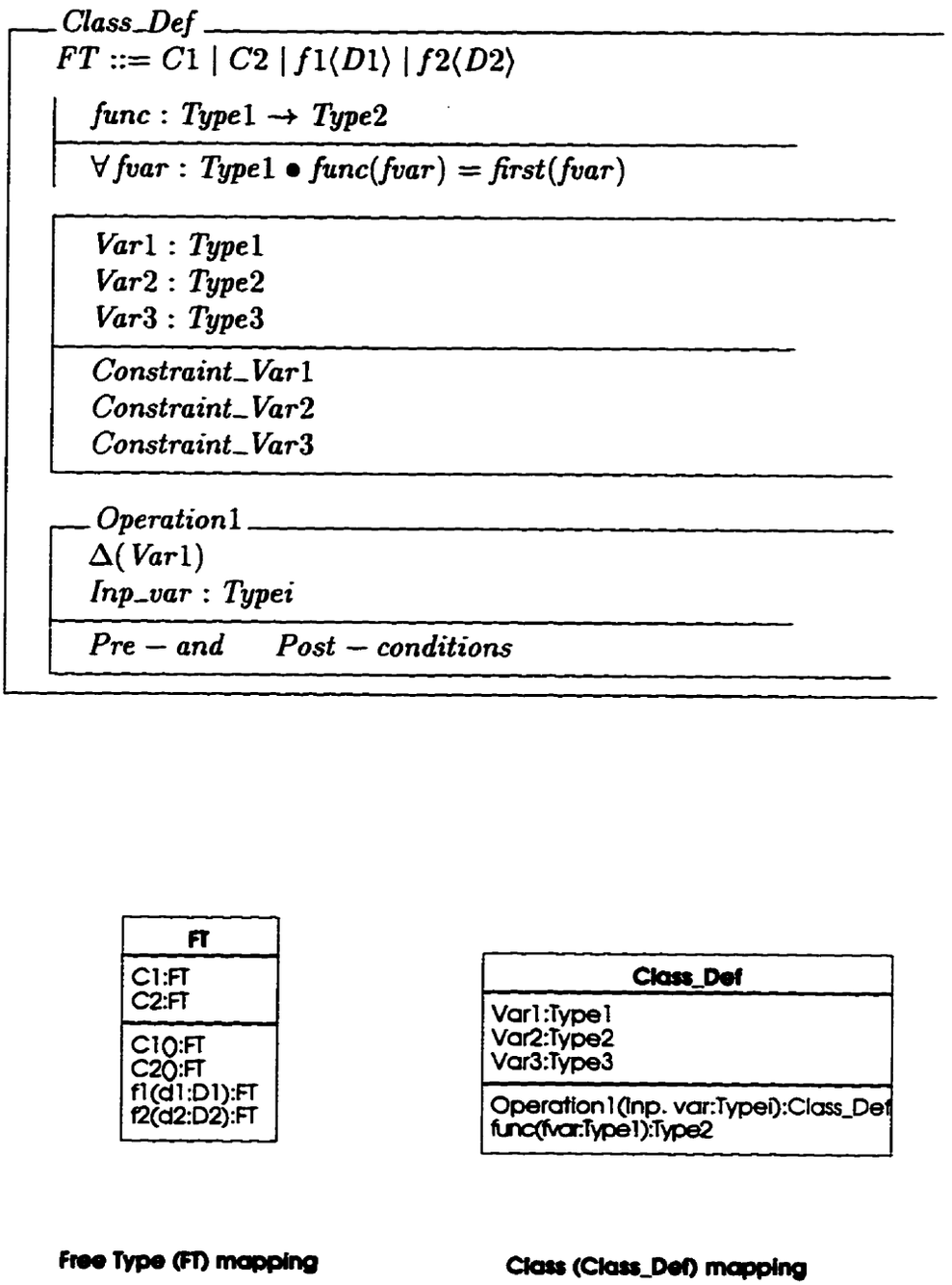


Figure 3.4: A Class in Object-Z and its design in OMT

Chapter 4

Test Case Generation Methodology

The existence of formal requirements specification for a system makes test case generation easier, since a requirements specification captures the intended functionality of a system at an abstract level. If a model-based specification language is used, the test case generation process has an added advantage of checking the implementation using its abstract model. Such a process more closely resembles the process of black-box testing of an application. The significant aspect in generating test cases from specification is to achieve a more complete coverage of the application. Since, we use a model-based specification language in conjunction with object-oriented approach (which supports a greater degree of modularity), we claim that our methodology would exhaustively cover more test cases than those methods described in [7, 22]. Though the proposed methodology uses an engineered discipline of deriving design

and implementation from the specification, testing the implementation is still necessary mainly because we did not use a strict formal refinement strategy. Moreover, the semantics of the programming language have not been addressed in the transformation process.

Before discussing about the proposed test case generation approach, we provide a semi-formalized notation of the design process. This notation is introduced to simply provide a more comprehensive understanding of the design phase, which contributes significantly to the test case generation process. We do not use any specification language, but we do use a collection of known symbols.

4.1 Formalization of Object Model

An object model (OM) in the object-oriented design of the application domain consists of a collection of classes. It also describes the relationships between classes.

Formally,

$$OM = (\mathbb{P}\, C, \mathbb{P}\, R)$$
$$\#C > 1 \Rightarrow \#R \geq 1$$

where $\mathbb{P}\, C$ represents a set of classes and $\mathbb{P}\, R$ the set of relationships that exist between the classes. The constraint indicates that an object model should contain at least two classes and one relationship; the symbol # indicates the cardinality of a set.

A class C, contains a set of attributes ($Attr$) and a set of operations (Opr) that operate on the attributes. For a class definition to be valid, we impose the constraint that both $Attr$ and Opr cannot both be empty simultaneously.

$$C = (\mathbb{P}\, Attr, \mathbb{P}\, Opr)$$
$$\neg\, (Attr = \{\} \wedge Opr = \{\})$$

Attributes ($Attr$) collectively contribute to object's state and operations (Opr) contribute to its behavior. In the object model, an attribute is defined by its name and type; its association with value domain is brought into existence only at run-time. We call the (name, type) part as the signature of the attribute.

$$Attr = Sign = (Name,\ Type)$$
$$Name = Identifier \qquad (*\ Userdefined\ *)$$
$$Type = \mathbb{P}\ Value$$

An operation (Opr) is identified by its unique name ($Name$), a set of parameters and the type of value it returns ($Type$).

$$Opr = (Name, \mathbb{P}\, Param,\ Type)$$
$$Param = Sign$$

We next define three kinds of relationships among the classes. They are i) *Association* ii) *Aggregation* and iii) *Inheritance.* Formally,

$$R = \{Association,\ Aggregation,\ Inheritance\}$$

We use the following simplified notations for the three relationships.

$C_1 \diamond\!\!-\!\!-\!\!-\, C_2 \equiv C_1 \quad$ *aggregates* $\quad C_2$

where C_2 is a component of C_1 (C_1 is the aggregate).

$C_1 \lhd\!\!-\!\!-\!\!-\, C_2 \equiv C_2 \quad$ *inherits* $\quad C_1$

where C_1 is the superclass and C_2 is the subclass. For inheritance by expansion, the following conditions must be true:

$C_1.Attr \subseteq C_2.Attr$
$C_1.Opr \subseteq C_2.Opr$

The last relation, *association*, denotes a semantic dependency between two classes in a cluster.

$C_1 \bullet\!\!-\!\!-\!\!-\!\!\rightarrow C_2 \equiv C_1 \quad$ *is − associated − with* $\quad C_2$

We assert that $C_1 \bullet\!\!-\!\!-\!\!-\!\!\rightarrow C_2$ denotes a unidirectional dependency in which C_1 is semantically dependent on C_2. Using Rumbaugh's [19] and Booch's [4] interpretation, there exists an attribute, a parameter to an operation or a local variable to an operation in C_1 which is of type C_2. Formally,

$(\exists\, a \in C_1.Attr \bullet a.Type = C_2) \wedge$
$(\exists\, op \in C_1.Opr \bullet (\exists\, p \in op.Param \bullet p.Type = C_2) \vee (op.Type = C_2))$

4.2 Formalization of Dynamic Model

A dynamic model (DM) consists of a collection of state transition diagrams. Each state diagram corresponds to exactly one class.

$$DM \equiv C \nrightarrow SD$$

where SD represents a state transition diagram. DM is represented as a partial function because there might be some classes for which we do not represent the state transition diagram.

A state diagram (SD) contains states and transitions.

$$SD \equiv (\mathbb{P}\, States, \mathbb{P}\, Trans)$$

To be meaningful and useful, a state transition diagram should contain at least two states and one transition. Therefore,

$$\#States > 1 \Rightarrow \#Trans >= 1$$

A transition is defined between a pair of states. It may be associated with a condition, a set of actions and an activity. The result of performing an activity may generate a set of events. These notions are formalized below.

$$Trans \equiv \{(State \times State), Condition, \mathbb{P}\, Param, \mathbb{P}\, Action, Effect\}$$

where

$$Condition = Predicate$$
$$Action = Opr$$
$$Effect = (Activity, \mathbb{P}\, Event)$$
$$Event = Predicate$$
$$Activity = Opr$$

A state is characterized by a collection of values for the attributes.

$$State = Name \rightarrow Value$$

One might extend the formal definitions further to show the relationship between the object model and dynamic model. This thesis does not elaborate the definitions any further, since it is beyond the scope of this thesis.

4.3 Test case for structural components

In this section, we provide test cases for validating the static entities in a system. Test cases for structural components (variable declarations) are generated by partitioning the value domain, for each type. Importance is given to boundary conditions, special cases and exceptional situations during partitioning. This renders special significance to the testing technique. For completely testing a particular component, test cases for all the partitions are unionized. For example, if

$$T_1, T_2, T_3, \ldots T_n$$

represent the test cases of the partitions, then test case for that component denoted by T, is collectively provided by,

$$T \cong T_1 \cup T_2 \cup T_3 \cup \ldots \cup T_n$$

We further elucidate all possible data types present in a specification and provide test cases for each of them.

4.3.1 Primitive data types

In this subsection, test cases for the four different primitive types, *natural number*, *natural number excluding zero*, *integer* and *real* are provided.

Natural Number

Any variable of this type can possess values from 0 through a *upper bound* value. This upper bound is designated by design or implementation constraints. There are four partitions to test a variable of this type.

$T_1 = \{ 0 \}$

$T_2 = \{ \text{upperbound} \}$

$T_3 = \{ \text{value} \mid 0 < \text{value} > \text{upperbound} \}$

$T_4 = \{ \text{value} \mid \text{value} > \text{upperbound} \}$

Except for the value T_4, values in all other partitions are expected to give correct results.

Natural Number excluding Zero

This case is similar to the previous type, but with the *lower bound* as 1.

$T_1 = \{ 1 \}$

$T_2 = \{ \text{upperbound} \}$

$T_3 = \{ \text{value} \mid 1 > \text{value} < \text{upperbound} \}$

$T_4 = \{ \text{value} \mid \text{value} > \text{upperbound} \}$

Integer

A variable of integer type can possess any value between a *lower bound* and an *upper bound.* Generally the design or implementation provides the boundary values.

T_1 = { value | value < lowerbound }

T_2 = { lowerbound }

T_3 = { 0 }

T_4 = { value | upperbound < value > lowerbound }

T_5 = { upperbound }

T_6 = { value | value > upperbound }

For values other than T_1 and T_6, correct results are expected for the rest of them.

Real

The Object-Z notation supports real type, which is not supported by its ancestor Z. Apart from taking the values as supported by an Integer type, it can also possess fractional values as well that lies in the prescribed range. Either the design and/or implementation must be responsible for restricting the range by providing a lower and an upper bound value.

T_1 = { value | value < lowerbound }

T_2 = { lowerbound }

T_3 = { 0 }

T_4 = { value | upperbound < value > lowerbound }

T_5 = { upperbound }

T_6 = { value | value > upperbound }

Even though the partitions for real values look similar to those for integer type, testing real values require more test cases because of the precision. Such precision dependencies are not obvious from the specification. We augment more test cases by selecting more values from T_1, T_4 and T_6 above.

4.3.2 User-defined types

Basic types in Z and Object-Z notations are assumed to be basic or primitive for the current specification and hence representational details of these basic types will not be available in the current specification. Such basic types are refined further during design and/or implementation. Since the basic types do not have any representations in the current specification, generating test cases for these types is obviously deferred.

In our approach, we look into the design elements corresponding to the basic types and generate test cases from the design. This is one of the occasions where information from the design is used for test case generation.

4.3.3 Composite data types

In this subsection, we deal with generating test cases for composite data types: *set types*, *cartesian product types* and *schema types*.

Set and Set types

A set can be expressed in three possible ways: *set as a type*, *set by enumeration* and *set by comprehension.*

Set as a type

When a variable is defined as a set type (e.g., x : $\mathbb{P}$ x), the variable can become equal to any member of the state; in other words, it may assume values enumerated by the set. Since testing every possible assignment of members from the set is almost impossible, we use a partitioned approach. That is, we consider two major partitions: i) an empty set; and ii) a non-empty set.

Moreover, informally it is agreed by the Z community that a set type in Z (or Object-Z) specification is finite. We follow a rigorous approach in mapping such a finite set to an array in the design. Consequently, an array has a bound. This bound enables us to split the second partition into two sub-partitions: i) a set with size smaller than allocated and ii) a set with size equal to the size of allocation. Thus, the valid test cases for a set type will be,

$T_1 = \{\ \}$	empty set
T_2 - $\{x\}$ mod $\#\{x\} = 1$	unary set
T_3 - $\{x\} \mid \#\{x\} < index$	index is the array size
T_4 - $\{x\} \mid \#\{x\} = index$	

Notice that $\mathbb{N}$, $\mathbb{N}_1$, $\mathbb{Z}$ and $\mathbb{R}$ are all sets; therefore, the test case generation process for set type is very similar to these predefined types, with one exception: the partition for the set if beyond the allocated size cannot be tested because such a test depends on the elements of the set. For numerals such as integers and reals, it is possible to generate a number outside a given range, but for an arbitrary set, it is almost impossible.

Set by enumeration

In this representation, data items in the set are enumerated. Since the set would be finite and small in this representation, it is possible to generate the test cases for each value explicitly. Thus,

$T_1 = \{ x_1 \}$ unary set

$T_2 = \{ x_1, x_2 \}$

$\vdots$

$T_n = \{ x_1, x_2, \ldots x_n \}$

Set by comprehension

This is yet another representation of a set which establishes the characteristics of individual elements that are present. It is impossible to generate test cases for all the members and hence partitioning criteria is essential.

For a set comprehension such as , $\{x : X \mid x \in D \wedge x \neq V\}$, where X is of a set type, D is some domain and V is a particular value in X, we generate the following partitions:

T_1 - { $value \mid value \in X;\ value \in D;\ value \neq V$ }

T_2 - { $value \mid value \in X;\ value \notin D;\ value \neq V$}

T_3 - { $value \mid value \in X;\ value \in D;\ value = V$}

T_4 - { $value \mid value \notin X;\ value \in D;\ value \neq V$ }

T_5 - { $value \mid value \in X;\ value \notin D;\ value = V$ }

T_6 - { $value \mid value \notin X;\ value \in D;\ value = V$ }

T_7 - { $value \mid value \notin X;\ value \notin D;\ value = V$}

T_8 - { $value \mid value \notin X;\ value \notin D;\ value \neq V$}

4.3.4 Tuples and Cartesian Product

Objects of different types can be grouped together as tuples to form cartesian product, which can be used to form new types. For example, a cartesian product called *Student_Info* can be formed using two other types *Name* and *ID* as follows:

$$Student\_Info = Name \times ID$$

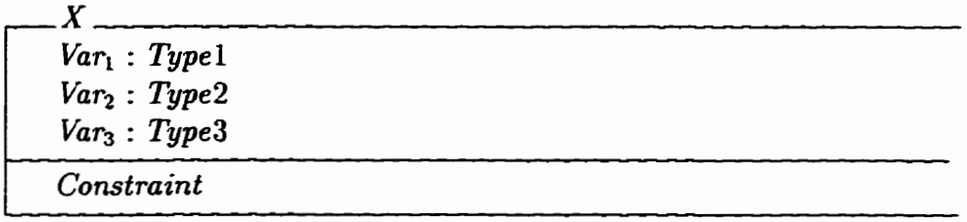
Since a cartesian product represents a collection of ordered n-tuples, we need to generate test cases for the ordering of elements in addition to test cases for individual elements.

For the above example, we have to first generate test cases for *Name* and *ID* which are sets; we then generate elements (n,i) where n is the name and i is the id; we also generate test cases for invalid pairs such as (n,n), (i,i), (i,n).

For a triple or quadruple, the number of test cases for ordering would still be finite.

4.3.5 Schema Types

A schema is a formal mathematical text which is used to divide the specification of a system into manageable chunks; it can also be used as a type. The variables in the declaration part of a schema forms the schema type. Test cases for schemas must also take care of the state invariants that have been specified in its predicate part. For example, consider the following schema :



In the schema X, there are three variables Var_1, Var_2 and Var_3 . Test case for schema X will be,

T_1 - { T_{Var_1}, T_{Var_2}, T_{Var_3} }

T_2 - { TN_{Var_1}, T_{Var_2}, T_{Var_3} }

T_3 - { T_{Var_1}, TN_{Var_2}, T_{Var_3} }

T_4 - { T_{Var_1}, T_{Var_2}, TN_{Var_3} }

T_5 - { TN_{Var_1}, TN_{Var_2}, T_{Var3} }

T_6 - { TN_{Var_1}, T_{Var_2}, TN_{Var_3}}

T_7 - { T_{Var_1}, TN_{Var_2}, TN_{Var_3}}

T_8 - { TN_{Var_1}, TN_{Var_2}, TN_{Var_3}}

In the above set of partitions, prefix T stands for conditions satisfying corresponding constraints for variables and TN stands for violations of them, if any. The number of test cases to test a schema type is proportional to the number of variables that are present.

4.3.6 Functions

In Z notation, functions are treated identically to relations. It includes *total*, *partial*, *injective*, *surjective*, and *bijective* functions.

Partial Function

For a partial function fp, $fp : X \pfun Y$, we have the following partitions:

$$T_1 = \{ x, y \mid x \in X \land y \in Y \land P(x, y) \}$$

$$T_2 = \{ x, y \mid x \in X \land y \in Y \land \neg (P(x, y)) \}$$

$$T_3 = \{ x, y \mid x \in X \land y \notin Y \land P(x, y) \}$$

$$T_4 = \{ x, y \mid x \in X \land y \notin Y \land \neg (P(x, y)) \}$$

$$T_5 = \{ x, y \mid x \notin X \land y \in Y \land P(x, y) \}$$

$$T_6 = \{ x, y \mid x \notin X \wedge y \in Y \wedge \neg (P(x, y)) \}$$

$$T_7 = \{ x, y \mid x \notin X \wedge y \notin Y \wedge P(x, y) \}$$

$$T_8 = \{ x, y \mid x \notin X \wedge y \notin Y \wedge \neg (P(x, y)) \}$$

where *P(x,y)* denotes a predicate on x and y. This predicate restricts the domain X to a subset for which the partial function is domained.

Total Function

The set of total functions from X to Y is a subset of the set of partial functions between X and Y. Hence, the test cases for a total function from X to Y should be a subset of the test cases for a partial function. The predicate *P(x,y)* in the definition of test cases for a partial function is set to true for a total function; the purpose of this predicate is to restrain a subset of the domain elements being not accepted as input. By setting *P(x,y)* to be true, every x in the domain is accepted, which is the characteristic of a total function. This effectively reduced the number of partitions of test cases for a total function to four.

Injective Function

An injective function can be partial or total. Therefore, partitions for partial and total functions described earlier can be adopted directly for an injective function. In addition, the injectivity of the function (namely, no two domain elements map to the same range element) must also be tested.

Exhaustive testing of injectivity requires that every pair of elements in the domain must be checked so that they map to distinct elements in the range. This is practically impossible. Since functions in Z (and in Object-Z) are finite, an implementation generally choses a bounded array to implement a function *. Therefore, such an exhaustive testing is still possible. In order to reduce the number of test cases in such occasions, one can use the application domain knowledge or some heuristic which is justifiable. For example, if the domain can be partitioned into disjoint subsets knowing that no two elements from different subsets will map into the same range element, then it is sufficient to test only pairs within each subset.

The additions partition required for a total injective function f_{IF},

$f_{IF} : X \rightarrowtail Y$, is

$T_{IF} = \{ x_1, x_2, y \mid x_1 \in X \wedge x_2 \in X \wedge y \in Y \wedge x_1 \neq x_2 \Rightarrow f_{IF}(x_1) \neq f_{IF}(x_2)\}$

and for a partial injective function f_{IPF},

$f_{IPF} : X \rightarrowtail\!\!\!\!\!\!\circ\ \ Y$, is

$T_{IF} = \{ x_1, x_2, y \mid x_1 \in X \wedge x_2 \in X \wedge y \in Y \wedge P(x_1, y) \wedge P(x_2, y) x_1 \neq x_2 \Rightarrow f_{IPF}(x_1) \neq f_{IPF}(x_2)\}$

*A function as a data type is implemented as an array whereas a function definition is implemented by a procedure or function; we are considering a function as a data type.

Surjective Function

Testing a surjective function is very similar to testing a total function because every element in the range of a surjective function is mapped by that function. However, finding the set of domain elements which map all elements in the range is practically difficult. We need application domain information or heuristic knowledge in this case, as we used in injective functions.

Bijective Function

According to Z (and Object-Z), a bijective function is a total injective and surjective function. Hence, the test cases for a bijective function would simply be the intersection of the partitions for the total injective and partitions for surjective functions.

4.3.7 Sequences

A sequence is an ordered collection of objects. Sequences in Z can be represented in two ways: *sequence as a type* and *sequence by enumeration.*

Sequence as a Type

The notation *seq X* denotes a sequence, where X is either a basic or composite type. As in the case of set type, we partition the test cases for a sequence based on the size of the sequence. Consequently,

T_1 - { x | $\#(seqX) = 0$ } empty sequence

T_2 - { x | $\#(seqX) = 1$ } unary sequence

T_3 - { x | $\#(seqX) < lowerbound$ }

T_4 - { x | $\#(seqX) = upperbound$}

Sequence by enumeration

Unlike sequence as a type, where the ordering among the elements in not obvious, sequence by enumeration explicitly refers to the elements and their ordering. Thus, if $\langle x, y, z, \ldots t\rangle$ is an enumerated sequence, the test cases would be

$T_1 = \langle\rangle$ empty sequence

$T_2 = \langle x\rangle$ unary sequence

$T_3 = \langle x, y\rangle$

$T_4 = \langle x, y, z\rangle$

$\vdots$

$T_n = \langle x, y, z, \ldots, t\rangle$

4.3.8 Free Types

A free type definition is used for defining enumerated constants and recursive data types. For example, a list will be defined as

$List :: nil \mid cons\langle\langle Element \times List\rangle\rangle$

The second part need not be recursive. As an example,

$User :: stud\langle\langle Student \rangle\rangle \mid fac\langle\langle Facult \rangle\rangle$ is a free type definition which asserts that a user can be a faculty user or a student user. An implicit constraint in a free type definition is that the carrier set of the free type is partitioned into disjoint subsets which are separated by the '|' symbol. Thus, in the definition of *" List"*, the constant *nil* will not be available in the range of the function *cons*. Further, all the functions appearing in a free type definition are totally injective.

Using these information, we can generate test cases for a free type definition as a set of disjoint subsets corresponding to the partitions in the free type definition.

4.4 Test cases for methods

In this section, we discuss test case generation from operations and functions in the specification. An operation or a function in the specification is defined by a set of assertions which collectively contains the preconditions and post-conditions. A precondition of an operation indicates the constraints that must be true before the operation is invoked. Obviously, it tests the validity of the environment in which the operation can be safely invoked. In a similar way, the postcondition of an operation asserts the constraints that must hold after the operation successfully terminates. Both pre and postconditions are

expressed in conjunctive normal form such as

$$preOp = Pred_1 \wedge Pred_2 \wedge \ldots \wedge Pred_k$$
$$postOp = Pred_{k+1} \wedge Pred_{k+2} \wedge \ldots Pred_n$$

It is therefore possible to generate test cases from each one of the predicates in the pre and postconditions. In our method, we generate test cases based on each one of the predicates (when it is true and when it is false).

4.5 Polymorphic Substitutions

In object-oriented programs polymorphism occurs when a variable name is bound to objects of different classes, one at a time. These classes must be related through inheritance. So, an operation expecting an object of a superclass can be substituted by an object belonging to any of its subclasses. This leads to additional test cases since all possible object substitutions must be taken care of.

The specification language, Object-Z supports only *inheritance by expansion.* Using this, specialized classes can only add new properties but cannot redefine any of the inherited properties. Classes in the inheritance hierarchy will necessarily include all the properties that have been defined in the classes which are present at one level higher in the hierarchy. If C_1 is a superclass and C_2 is one of its subclasses then,

$$C_1.Attr \subseteq C_2.Attr$$
$$C_1.Opr \subseteq C_2.Opr$$

Hence the leaf classes in the hierarchy will eventually contain all the information from the class(es) which are present above them. In such a situation, it is enough if the leaf classes are tested.

When *inheritance by redefinition* is permitted, we can still make use of the test cases for the subclass that retain the unmodified features of the superclass. This partial reuse of test cases has not been included in this thesis; it has been reserved as one of the continuing work of this research.

4.6 An example

We now illustrate the test case generation methodology for a simple automated teller machine. Following is the Object-Z class specification for the *Bank*: a more detailed description of the example, the specification and test cases are given in the Appendix.

Bank

$$
\begin{array}{l}
users : \mathbb{P}\, User \\
accounts : \mathbb{P}\, Account \\
money\_pool : \mathbb{R} \\
\hline
money\_pool \geq 0 \\
\bigcup\{u : User \mid u \in users \bullet u.accounts\} = \\
\qquad \{a : Account \mid a \in accounts \bullet a.number\} \\
\{a : Account \mid a \in accounts \bullet a.owner\} = \\
\qquad \{u : User \mid u \in users \bullet u.id\} \\
\forall u : users \bullet (u \in General \lor u \in Privileged \lor u \in Staff) \\
\forall a : accounts \bullet a \in Savings \lor a \in Chequing \lor a \in FixedDeposit
\end{array}
$$

INIT

$$
\begin{array}{l}
users = \varnothing \\
accounts = \varnothing \\
money\_pool > 0
\end{array}
$$

Bank(Contd...)

OpenAccount$_0$

$$
\begin{array}{l}
\Delta(users, accounts) \\
new? :\downarrow Account \\
user? :\downarrow User \\
\hline
user? \in users \\
new?.number \notin \{a : Account \mid a \in accounts\} \\
accounts' = accounts \cup \{new?\}
\end{array}
$$

$$
\begin{array}{l}
OpenAccount \mathrel{\widehat=} OpenAccount_0 \bullet \\
\qquad user?.AddAccount[new?.number/new\_account?]
\end{array}
$$

The specification for *User* and *Account* are given in the appendix. The state invariants ensure consistency between user records and accounts.

4.6.1 Structural validity

As stated in the methodology, initially test cases will be generated to validate the static entities. The variables present in the class Bank are validated.

users

As *users* is defined as a set, the following partitions of test cases are generated for *users*.

T_1 - #users = 0

T_2 - #users = 1

T_3 - #users > 1

T_4 - #users = Upper index

T_5 - #users > Upper index

Notice that the *Upper index* must be defined either in the design or at the implementation.

accounts

The variable *accounts* is also defined as a set. This leads to the following partitions.

T_1 - #accounts = 0

T_2 - #accounts = 1

T_3 - #accounts > 1

T_4 - #accounts = Upper index

T_5 - #accounts > Upper index

The value for *Upper index* for *accounts* may be different from that used for *users*.

money_pool

This is defined as a variable of type *Real*. So, the partitions will correspond to boundary cases designated by the upper and lower limits set by the implementation. However, it cannot be negative. So, the lower value is 0.

$T\_1 = \{ 0 \}$

$T\_2 = \{$ value | lower bound > value < upper bound $\}$

$T\_3 = \{$ upper bound $\}$

$T\_4 = \{$ value | value > upper bound $\}$

4.6.2 Validating OpenAccount_0

Static Validity of *new* :↓ *Account*

For Savings type of account

This is simply an *Account* class without any addition of properties. The test cases for this are comprised of those corresponding to the components of the *Account* class, which are shown below:

balance

$T_1 = \{ 0 \}$

T_2 = { upper bound }

T_3 = { value | 0 > value < upper bound }

T_4 = { value | value > upper bound }

interest_rate

T_1 = { 0 }

T_2 = { upper bound }

T_3 = { value | 0 > value < upper bound }

T_4 = { value | value > upper bound }

interest_calculated

Test sets follow constraints specified in design and/or implementation strategies for this variable.

owner

T_1 = { value | value < lower bound }

T_2 = { lower bound }

T_3 = { value | lower bound > value < upper bound }

T_4 = { upper bound }

T_5 = { value | value > upper bound }

For Chequing type of account

This makes use of all the test cases defined for *Account* (and hence *Savings Account*) defined earlier; it also includes test cases for the additional variable; *minimum_balance*.

minimum_balance

T_1 = { min_bal }

T_2 = { upper bound }

T_3 = { value | min_bal > value < upperbound }

Value for *min_bal* must have been set during design or implementation.

For Fixed_Deposit type of account

The two additional variables for this are *start* and *end* which indicate the duration of the fixed deposit. Test cases based on these two variables are given below:

start

Test data depends on constraints stated in design and/or implementation for *Date* data type.

end

Test data depends on constraints stated in design and/or implementation for *Date* data type.

Static Validity of *user*? :↓ *User*

From the object model of ATM, it can be inferred that the substitutable classes does not add any new properties. Hence, test cases will be derived only for the specialized class in the hierarchy i.e, *User*. The test cases are as follows:

id

T_1 = { lowerbound }

T_2 = { 0 }

T_3 = { value | lowerbound > value < upperbound }

T_4 = { upperbound }

T_5 = { value | value > upperbound }

accounts

T_1 = # accounts = 0

T_2 = # accounts = 1

T_3 = # accounts = upper bound

T_4 = # accounts > upper bound

Operational validity

After ensuring that static properties is in conformance to their formulated requirements, we now generate test cases for operations. Here we show only the test cases for the operation *OpenAccount.*

The following predicates are identified as candidates for test case generation:

1. user? $\in$ users new?.number $\notin$ accounts
2. user? $\in$ users new?.number $\in$ accounts
3. user? $\notin$ users new?.number $\notin$ accounts
4. user? $\notin$ users new?.number $\in$ accounts

Using the formulated test cases, operation *Open_Account* can be validated. Exercising these data sets will assure confidence in the functionality of this

operation. It must be noted that this operation fires another transition in *Account* class by enabling through event synchronization which will be taken care of by the test cases formed in the *Account* class.

Naturally by testing the transitions that can be invoked with the help of the state transition diagrams, the system under consideration can be validated against its requirements.

Chapter 5

Observation

The proposed method was successfully applied to two different case studies: a *library management system* and an *automated teller machine* application. In this chapter, the results which were inferred from those case studies are discussed. Details about the application of this method for those case studies has been explained in the appendices. Both the case studies are fairly complex and large. The case studies were engineered through the software development life cycle. As test cases generated using this method are used to validate a software against its requirements specification, implementation was performed using both *C++* [23] and *Smalltalk* [14]. The programs were written in such a way that they utilize polymorphic substitution of objects at run-time. Finally, test cases deduced from the notation taking the design information into consideration were applied on them. The test cases are independent of the semantics of implementation. Both implementations were validated successfully against their formulated requirements.

In addition to generating test cases to perform validation, the method also demonstrates a systematic approach to software development, from requirements specification through testing. This shows the importance of formal methods not only in the requirements analysis but even to the other phases in the software life cycle. But like any other specification-based test case generation method the success of this method largely depends upon the correctness of the specification, which is still under research.

The design process is fairly simple because of the straight forward justification of relationships in the OMT. This is in contrast to deriving an object-oriented design from the informal requirements of a system which is tedious and often involves iterative development. When deriving design from the specification, the transitions, activities and actions can be easily identified from the operations. In addition, the state change information which has been captured in the specification for operations clearly illustrates the variables that are affected by the operations. This is of tremendous use both in design as well as in the implementation phase.

Though not used in this thesis, the semi-formal description of the test case generation process will make process easily automatable. This, along with the requirements specification, helps justifying the reduction in the number of test cases that have to be considered under polymorphic substitutions. And further, it also demonstrates the soundness of this testing method.

One of the disadvantages with any testing technique, in general is the number of the test cases that must be generated for true validation. This becomes worse in situations such as polymorphic substitutions in object-oriented

programs. The testing process becomes more tedious particularly if the inheritance hierarchy is high. In this thesis, this issue is addressed when the inheritance is by *expansion*. The situation which arises when the inheritance is by *redefinition* is not probed in detail. The reason being partially the inadequacy of Object-Z notation to specify it. When the inheritance is by expansion , the subclass inherits all of the superclass's features without modifying them. In such a situation, the use of formal specification as a basis in generating test cases justifies the reuse of test cases for subclasses for testing superclass. The reusability of test cases also leads to reduction in the number of test cases.

5.1 Comparison with other existing methodologies

The approach for testing a system suggested in this thesis is quite novel. The notion of deriving test cases from requirements specification and using relevant design information makes the method more concrete. It also covers almost all possible situations. In this section, the effectiveness of this testing method is analyzed by comparing with other existing approaches. The significant contribution of this thesis is henceforth highlighted.

Among the gamut of testing techniques which are widely known, we compare our approach with some of them. The methodology developed by Turner and others [24] is based on the semantics of the implementation language. This work also provides a higher level guidance to testing object-oriented programs.

Reusability of test cases is achieved by classifying the features but it lacks any explanation for effectively categorizing it. Further, it does not discuss as to how any modifications in the code will affect the formulated test cases. It fails to address the much prevalent issue in object-oriented paradigm, namely polymorphism. Another program based testing technique proposed by McGregor and others [15] addresses polymorphic substitutions. But any changes to the software requirements cannot be easily accommodated using this approach. Further, it does not justify the sufficiency of the number of test cases due to polymorphic substitutions. As compared to this, a specification-based testing technique suggested by Stocks [22] has dealt with the adaptation of some existing testing techniques to suit test case generation from formal requirements specification. Though the method offers a simple and elegant way of structuring test data, it does not provide any guideline for applying test cases to ensure the correctness of the system, in general.

As opposed to this, the test technique proposed in this thesis has the following significant aspects.

- The formal requirements specification and the object-oriented design in OMT both provide a clear understanding of the application.
- The thesis informally provide some guidelines for deriving design from the specification, which is of immense use as supposed to deriving the same from informal requirements.
- The design model in conjunction with the specification gives more test cases incorporating additional information added during the design stage.

This has an added advantage as compared to other specification-based approaches which do not include design information.

- Modifications in the requirements can be easily accommodated in the test data by tracing the effect on other existing test cases. Though we do not have any procedure to justify the tracing process, our experience shows that it is easier in this approach. This is because the proposed method also includes a systematic development of object-oriented programs.

- The use of formal specifications as a basis in generating test cases makes it feasible to justify the minimality of the number of test cases in the presence of polymorphism.

- In our approach, the static components are validated using state invariants that are present in the specification. These are very important as they contribute to a well-defined set of test cases.

- The thesis also includes a semi-formal description of the method which, we hope, will help in automating the process in the future.

Bibliography

[1] S. Barbey, M.M. Ammann, and A. Strohmeier. Open issues in testing object-oriented software. In *Proceedings of the European Conference on Software Quality(ECSQ94)*, pages 257–267, Basel, Switzerland, 1994.

[2] S. Barbey, D. Buchs, and C. Peraire. A theory of specification-based testing for object-oriented software. Technical Report 96/163, Swiss Federal Institute of Technology, Computer Science Dept., Software Engineering Laboratory, Jan 1996.

[3] G. Bernot, M.C Gaudel, and B. Marre. Software testing based on formal specifications: A theory and a tool. *Software Engineering Journal (UK)*, 6(6):387–405, Nov 1991.

[4] G. Booch. *ObjectOriented analysis and design with applications.* Benjamin/Cummings Publishing Company, 1994.

[5] G. Booch et al. *The Unified Modeling Language for Object Oriented Development: Documentation Set Version 0.8.* Rational Software Corporation, 1995.

[6] G. Booch et al. *The Unified Modeling Language for Object Oriented Development: Documentation Set Version 0.91 Addendum.* Rational Software Corporation, 1996.

[7] D. Carrington and P. Stocks. A tale of two paradigms: Formal methods and software testing. Technical Report 94-4, Software Verification Research Centre, The University of Queensland, Feb 1994.

[8] B. Forghani and B. Sarikaya. Semi-automatic test suite generation from estelle. *Software Engineering Journal (UK)*, 7(4):295–307, July 1992.

[9] M.J. Harrold, J.D. McGregor, and K.J. Fitzpatrick. Incremental testing of object-oriented class structures. In *Proceedings of the International Conference on Software Engineering*, pages 68–80, 1992.

[10] D.M. Hoffman and P.A. Strooper. Graph-based class testing. *The Australian Computer Journal*, 26(4):158–163, Nov 1994.

[11] D.M. Hoffman and P.A. Strooper. The testgraphs methodology: Automated testing of collection of classes. *Journal of Object-Oriented Programming*, pages 30–36, Nov/Dec 1995.

[12] S. Kirani. *Specification and Verification of Object-Oriented Prog rams.* PhD thesis, Dept. of Computer Science, University of Minnesota, Nov 1994.

[13] T. Korson and J. McGregor. Understanding object-oriented: A unifying paradigm. *Communications of the ACM*, 33(09):40–60, Sep 1990.

[14] W. Lalonde. *Discovering Smalltalk.* Benjamin/Cummings Publishings Company, 1994.

[15] R. McDaniel and J. McGregor. Testing the polymorphic interactions between classes. Technical Report TR94-103, Dept. of Computer Science, Clemson University,, 1995.

[16] A. Paradkar. Inter-class testing of o-o software in the presence of polymorphism. In *Proceedings of CASCON 1996*, pages 137–146, Nov. 1996.

[17] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z.* Prentice Hall International Ltd, 1991.

[18] G. Rose R. Duke and G. Smith. Object Z : A Specification Language Advocated for the Description of Standards. Technical Report 94-45, Software Verification Research Centre, The University of Queensland, Dec 1994.

[19] J. Rumbaugh. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, 1991.

[20] I. Sommerville. *Software Engineering-Fourth Edition.* Addison-Wesley, 1992.

[21] J. Spivey. *The Z notation : A Reference Manual (Second Edition).* Prentice Hall International Series in Computer Science, 1992.

[22] P.A. Stocks. *Applying formal methods to software testing.* PhD thesis, Department of Computer Science, University of Queensland, Dec 1993.

[23] B. Stroustrup. *The C++ programming language.* Addison-Wesley, 1991.

[24] C.D. Turner and D.J. Robson. The state-based testing of object-oriented programs. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 302–311, Sep. 1993.

Appendix A

Library Management System

A.1 Problem Description

An object-oriented specification of a *library management system* written in Object-Z notation is provided. Current version of this specification has not been type-checked.

The following are the assumptions and requirements for the library management system being specified:

1. There are three different types of items for loan in the library: *books*, *journals* and *permanently reserved books*. Permanently reserved books are similar to books except that they can be borrowed for only a few hours. All the three items are grouped together under one name called *loan item*. The specification is written in such a manner that the three types of items are polymorphic to *loan item*. Hence, most of the high level

operations are written only for *loan item* which will be polymorphically invoked for a particular type of item.

2. There are three types of *users* or *borrowers* (both terms being interchangably used in this report): *undergraduate students*, *graduate students* and *faculty members*. The borrowing privileges such as the number of loan items that can be borrowed and the due date or due time for a loan item vary depending on the type of the user. Some constants are used to indicate the differences in terms of the borrowing privileges but these can be changed in the implementation.

3. A user can borrow an item, return a loaned item or reserve an item.

4. The following restrictions apply for reserving items:

   - Permanently reserved books cannot be reserved by any user.
   - An undergraduate student cannot reserve a journal.
   - An item available on the shelf cannot be reserved.
   - A user cannot reserve an item more than once at a time.

5. A loan item has four distinct status:

   available which indicates that the item is available on the shelf and can be borrowed;

   loaned which indicates that the item is currently loaned to some user but is not reserved;

reserved which indicates that the item is on the shelf but some user(s) has/have reserved it;

reserved&loaned which indicates that the item has been loaned to some user and has also been reserved by some user(s).

6. The library working hours are fixed such as 9:00 A.M. to 8:00 P.M. Any permanently reserved item which is borrowed close to the closing hours of the library will have its due time at the closing hour even if its original loan period is longer. For example, if a user is entitled to borrow a permanently reserved item for 3 hours but the user is borrowing it at 7:00 P.M., then the user must return it at 8:00 P.M., the closing hour. This is not specified in the case study, but has been included for the sake of an understandable description.

A.2 Global Definitions

This section includes several type definitions and constants which are specified in Z. These definitions are global and can be shared by all class definitions that are to be described in the next section. We try as much as possible to complete the specification of these global definitions; any omissions here should be taken care of in the implementation.

$[CallNumber, STRING, UserID]$

$LoanItemStatus == \{Available, Reserved, Loaned, Reserved\&Loaned\}$

$VolumeNumber == 1 \,..\, 100$

$IssueNumber == 1 \,..\, 24$

$Year == 1900 \,..\, 2000$

$Month == 1 \,..\, 12$

$Day == 1 \,..\, 31$

$Hour == 0 \,..\, 23$

$Minute == 0 \,..\, 59$

$Time == Hour \times Minute$

$Date == Day \times (Month \times Year)$

The following projection functions are used to extract components of *Time*.

$$
\begin{array}{|l}
hour : Time \rightarrow Hour \\
minute : Time \rightarrow Minute \\
\hline
\forall\, t : Time \bullet hour(t) = first(t) \wedge minute(t) = second(t) \\
\hline
\end{array}
$$

The following projection functions are used to extract components of *Date*.

$$
\begin{array}{|l}
day : Date \rightarrow Day \\
month : Date \rightarrow Month \\
year : Date \rightarrow Year \\
\hline
\forall\, dt : Date \bullet \\
\quad day(dt) = first(dt) \wedge month(dt) = first(second(dt)) \wedge \\
\quad year(dt) = second(second(dt)) \\
\hline
\end{array}
$$

The following global constraint is applied to every instance of *Date* for validation.

$$
\begin{array}{l}
\forall\, d : Day;\ m : Month;\ y : Year \bullet (d, (m, y)) \in Date \Rightarrow \\
\quad ((m = 1 \vee m = 3 \vee m = 5 \vee m = 7 \vee m = 8 \vee m = 10 \vee m = 12) \\
\qquad \Rightarrow 0 \leq d \leq 31) \wedge \\
\quad ((m = 4 \vee m = 6 \vee m = 9 \vee m = 11) \Rightarrow 0 \leq d \leq 30) \wedge \\
\quad ((m = 2 \wedge year \bmod 4 = 0 \wedge year \bmod 100 \neq 0) \Rightarrow 0 \leq d \leq 29) \wedge \\
\quad ((m = 2 \wedge year \bmod 4 \neq 0 \vee year \bmod 100 = 0) \Rightarrow 0 \leq d \leq 28)
\end{array}
$$

The following constants define the *current date* and *current time.*

$$current\_date : Date$$
$$current\_time : Time$$

A.3 Class Definitions

We start with a generic class definition, *LoanItem.* A loan item contains a *title*, *year of publication* and a unique *call number.* These three attributes are common for every loan item such as a book, and a journal. In addition, a loan item has a *status* indicating whether it is in the library and is available for borrowing, in the library but has been reserved or it is loaned to someone.

LoanItem

$$title : STRING$$
$$year : Year$$
$$callno : CallNumber$$
$$status : LoanItemStatus$$

ChangeStatus

$$\Delta(status)$$
$$newstatus? : LoanItemStatus$$

$$status' = newstatus?$$

A *book* is a loan item with additional attributes: *authors* and *publisher.*

Book

$$LoanItem$$

$$authors : STRING$$
$$publisher : STRING$$

A *permanently reserved book* is a book which can be borrowed only for a few hours as opposed to borrowing a regular book for several days. It has an additional attribute indicating the number of hours (a maximum of 6 hours) for which the book can be borrowed. A permanently reserved book cannot be reserved by a user.

PermReserveBook

Book

$$loanperiod : \mathbb{N}_1$$

$$loanperiod < 6$$

A *journal* is another type of loan item which contains *volume* and *issue numbers*.

Journal

LoanItem

$$volumno : VolumeNumber$$
$$issuno : IssueNumber$$

A *user* of the library (also called a *borrower*) has a unique *id number*. The number of loan items a user can borrow varies with the type/category of the user.

User

$$idnumber : UserID$$
$$maxbooks : \mathbb{N}_1$$
$$maxjournals : \mathbb{N}_1$$
$$maxpermresbooks : \mathbb{N}_1$$

The three types of users in the library system are given below. The distinguishing feature is the number of loan items each user can borrow.

Ugrad

User

$year\_joined : Year$

$current\_year - year\_joined < 4$
$maxbooks = 10$
$maxjournals = 10$
$maxpermresbooks = 3$

An undergraduate student can use the library only for a maximum of four years. A graduate student can use the library for a maximum of three years.

Grad

User

$year\_joined : Year$

$current\_year - year\_joined < 3$
$maxbooks = 20$
$maxjournals = 20$
$maxpermresbooks = 5$

A faculty member has no restriction to use the library except for the maximum number of loan items that can be borrowed.

Faculty

User

$maxbooks = 40$
$maxjournals = 50$
$maxpermresbooks = 10$

A *loan record* is defined by a separate class definition which maps one user to one loan item. It also contains the date of borrow, the date to be returned, the time of borrow (if it is a permanently reserved book) and the time to be returned.

LoanRecord

$item :\downarrow LoanItem$
$borrower :\downarrow User$
$date\_borrowed : Date$
$date\_tobe\_returned : Date$
$time\_borrowed : Time$
$time\_tobe\_returned : Time$

Borrow

$\Delta(date\_borrowed, date\_tobe\_returned)$

$date\_borrowed' = current\_date$
$date\_tobe\_returned' =$
if $borrower \in Ugrad$ **then** $current\_date + 30$
else if $borrower \in Grad$ **then** $current\_date + 60$
else $current\_date + 120$

BorrowPermRes

$\Delta(time\_borrowed, time\_tobe\_returned)$

$time\_borrowed' = current\_time$
$time\_tobe\_returned' =$
if $borrower \in Ugrad$ **then** $current\_time + 1$
else if $borrower \in grad$ **then** $current\_time + 4$
else $current\_time + 6$

A *reservation record* contains a queue of user ids for each item that is reserved. A permanently reserved item cannot be reserved. in addition, undergraduate students cannot reserve journals. No user can reserve the same material twice.

ReserveRecord

$$item :\downarrow LoanItem$$
$$borrowers :\downarrow \text{iseq } User$$

$$item \notin PermResBooks$$
$$item \in Journal \Rightarrow \text{ran } borrowers \cap Ugrad = \varnothing$$

AddUser

$$\Delta(borrowers)$$
$$new? :\downarrow User$$

$$new? \notin \text{ran } borrowers$$
$$borrowers' = borrowers \frown \langle new? \rangle$$

DeleteUser

$$\Delta(borrowers)$$
$$old? :\downarrow User$$

$$\text{ran } borrowers = \text{ran } borrowers' \cup \{old?\}$$

We now model the library. A library contains a set of users, a set of loan items, a set of loan records and a set of reservation records. The state invariant asserts several properties of the library including (i) all call numbers are unique; (ii) all user ids are unique; and (iii) loan and reservation records must be consistent with the information available in the library (i.e., the items that are loaned and that are reserved should belong to the library and the users must have registered with the library).

Library

$items :\downarrow \mathbb{P}\ LoanItem$
$borrowers :\downarrow \mathbb{P}\ User$
$loans : \mathbb{P}\ LoanRecord$
$reserves : \mathbb{P}\ ReserveRecord$

$\forall it_1, it_2 : items \bullet it_1.callno = it_2.callno \Rightarrow it_1 = it_2$
$\forall u_1, u_2 : borrowers \bullet u_1.idnumber = u_2.idnumber \Rightarrow u_1 = u_2$
$\forall ln : loans \bullet$
$\quad (ln.item \in items \land ln.borrower \in borrowers)$
$\forall rs : reserves \bullet$
$\quad (rs.item \in items \land \mathrm{ran}\ rs.borrowers \subseteq borrowers)$
$\forall it : items \bullet$
$\quad (it \notin \{ln : loans \bullet ln.item\} \land it \notin \{rs : reserves \bullet rs.item\} \Rightarrow$
$\quad\quad it.status = Available) \land$
$\quad (it \in \{ln : loans \bullet ln.item\} \land it \notin \{rs : reserves \bullet rs.item\} \Rightarrow$
$\quad\quad it.status = Loaned) \land$
$\quad (it \notin \{ln : loans \bullet ln.item\} \land it \in \{rs : reserves \bullet rs.item\} \Rightarrow$
$\quad\quad it.status = Reserved)$

INIT

$items = \varnothing$
$borrowers = \varnothing$

Adduser

$\Delta(borrowers)$
$new? :\downarrow User$

$new?.idnumber \notin \{u : borrowers \bullet u.idnumber\}$
$borrowers' = borrowers \cup \{new?\}$

AddItem

$\Delta(items)$
$new? :\downarrow LoanItem$

$new?.callno \notin \{u : items \bullet u.callno\}$
$items' = items \cup \{new?\}$

Library(Contd...)

Borrow

$\Delta(loans, reserves)$
$user? :\downarrow User$
$item? :\downarrow LoanItem$

$user? \in borrowers$
$item? \in items$
(**let** $bs == \#\{ln : loans \mid ln.borrower = user? \land ln.item \in Book\}$
$\land\ js == \#\{ln : loans \mid ln.borrower = user? \land ln.item \in Journal\}$
$\land\ ps == \#\{ln : loans \mid ln.borrower = user? \land$
$ln.item \in PermResBooks\} \bullet item? \in Book \Rightarrow bs < user?.maxbooks$
$\land\ item? \in Journal \Rightarrow bs < user?.maxjournals \land$
$item? \in PermResBooks \Rightarrow bs < user?.maxpermresbooks)$
$item?.status = Available \Rightarrow$
$([\, stat : LoanItemStatus \mid stat = Loaned\,] \bullet$
$item?.ChangeStatus[stat/newstatus?]) \land$
$loans' = loans \cup \{\mu\, newln : LoanRecord \mid newln.item = item? \land$
$newln.borrower = user? \land newln.Borrow \bullet newln\}$
$item?.status = Reserved \Rightarrow$
$(\exists\, rs : reserves \mid rs.item = item? \land head\ rs.borrowers = user? \bullet$
$tail\ rs.borrowers = \varnothing \Rightarrow$
$([\, stat : LoanItemStatus \mid stat = Loaned\,] \bullet$
$item?.ChangeStatus[stat/newstatus?]) \land$
$loans' = loans \cup \{\mu\, newln : LoanRecord \mid$
$newln.item = item? \land newln.borrower = user? \land$
$newln.Borrow \bullet newln\} \land$
$reserves' = reserves \setminus \{rs\}$
$tail\ rs.borrowers \neq \varnothing\,] \Rightarrow$
$([\, stat : LoanItemStatus \mid stat = Reserved\&Loaned\,] \bullet$
$item?.ChangeStatus[stat/newstatus?]) \land$
$loans' = loans \cup \{\mu\, newln : LoanRecord \mid$
$newln.item = item? \land newln.borrower = user? \land$
$newln.Borrow \bullet newln\} \land$
$reserves' = reserves \setminus \{rs\}\,]$
$\land\ rs.Deleteuser)$

Library(Contd...)

Return

$\Delta(loans, items)$
$item? :\downarrow LoanItem$

$item? \in \{ln : loans \mid ln.item = item?\}$
$\exists\, rs : reserves \mid rs.item = item? \bullet$
ran $rs.borrowers \neq \varnothing \Rightarrow$
$([\, stat : LoanItemStatus \mid stat = Reserved\,] \bullet$
$item?.ChangeStatus[stat/newstatus?])$
ran $rs.borrowers = \varnothing \Rightarrow$
$([\, stat : LoanItemStatus \mid stat = Available\,] \bullet$
$item?.ChangeStatus[stat/newstatus?])$

Reserve

$\Delta(reserves)$
$item? :\downarrow LoanItem$
$user? :\downarrow User$

$item? \in items \wedge user? \in borrowers$
$item? \notin PermResBook$
$item? \in Journal \Rightarrow user? \notin Ugrad$
$item?.status \neq Available$
$item?.status = Loaned \Rightarrow$
$([\, stat : LoanItemStatus \mid stat = Reserved\&Loaned\,] \bullet$
$item?.ChangeStatus[stat/newstatus?])$
$(\exists\, rs : reserves \bullet rs.item = item? \Rightarrow$
$reserves' = reserves \setminus \{rs\} \cup \{\mu\, newrs : ReserveRecord \mid$
$newrs.item = item? \wedge$
$newrs.borrowers = rs.borrowers \frown \langle user? \rangle\})$
$\neg\, (\exists\, rs : reserves \mid rs.item = item?) \Rightarrow$
$reserves' = reserves \cup \{\mu\, newrs : ReserveRecord \mid$
$newrs.item = item? \wedge newrs.borrowers = \langle user? \rangle\}$

The operations *INIT*, *AddUser* and *AddItem* are self-explanatory. The operation *Borrow* describes the situation of borrowing an item from the library. It asserts the following: (i) validate the user and the item; (ii) check for the

limit on the number of items that can be borrowed; (iii) change the status of the item appropriately; (iv) check whether the item has been reserved, and if so, check whether the user is the first person in the reservation queue; and (v) update loans records and reservation records accordingly. The operations *Return* and *Reserve* are similar to *Borrow* and are self-explanatory.

In developing an object-oriented design of library system, construction of object and dynamic models play a predominant role. The following models portray static as well as dynamic properties associated with the system that is being modeled.

A.4 Mapping of Types

In this section, we provide a mapping of types in the specification to their corresponding implementation data types.

A.4.1 Mapping of Global Definitions

Type	Mapped Type	Constraint on the values
CallNumber	User Defined	Ref. implementation and/or design
STRING	User Defined	Ref. implementation and/or design
UserID	User Defined	Ref. implementation and/or design
LoanItemStatus	Enumerated	Must be any one of
		Available, Reserved, Loaned
		or Reserved & Loaned
VolumeNumber	Integer	Lowerbound 1
		Upperbound 100
IssueNumber	Integer	Lowerbound 1
		Upperbound 24
Year	Integer	Lowerbound 1900
		Upperbound 2000
Month	Integer	Lowerbound 1
		Upperbound 12
Day	Integer	Lowerbound 1

contd... on next page

contd... from previous page		
Type	Mapped Type	Constraints on the values
		Upperbound 31
Hour	Integer	Lowerbound 0
		Upperbound 23
Minute	Integer	Lowerbound 0
		Upperbound 59

A.4.2 Mapping of Class Definitions

Book

Name	Type	Constraint on the values
title	STRING	Ref. from design or implementation
year	Year	Ref. mapping from global definitions
callno	CallNumber	Ref. mapping from global definitions
status	LoanItemStatus	Ref. mapping from global definitions
authors	STRING	Ref. from design or implementation
publisher	STRING	Ref. from design or implementation

PermReserveBook

Name	Type	Constraint on the values
title	STRING	Ref. from design or implementation
year	Year	Ref. mapping from global definitions
callno	CallNumber	Ref. mapping from global definitions
status	LoanItemStatus	Ref. mapping from global definitions
authors	STRING	Ref. from design or implementation
publisher	STRING	Ref. from design or implementation
loanperiod	Integer	upperbound < 6

Journal

Name	Type	Constraint on the values
title	STRING	Ref. from design or implementation
year	Year	Ref. mapping from global definitions
callno	CallNumber	Ref. mapping from global definitions
status	LoanItemStatus	Ref. mapping from global definitions
volumno	VolumeNumber	Ref. mapping from global definitions
issuno	IssueNumber	Ref. mapping from global definitions

User

Name	Type	Constraint on the values
idnumber	UserID	Ref. mapping from global definitions
maxbooks	Natural Num. excl. 0	Ref. from design or implementation
maxjournals	Natural Num. excl. 0	Ref. from design or implementation
maxpermreservebooks	Natural Num. excl. 0	Ref. from design or implementation

Ugrad

Name	Type	Constraint on the values
idnumber	UserID	Ref. mapping from global definitions
maxbooks	Natural Num. excl. 0	upperbound <= 10
maxjournals	Natural Num. excl. 0	upper bound <= 10
maxpermreservebooks	Natural Num. excl. 0	upper bound <= 3
year_joined	Year	current_year-year_joined < 4

Grad

Name	Type	Constraint on the values
idnumber	UserID	Ref. mapping from global definitions
maxbooks	Natural Num. excl. 0	upper bound <= 20
maxjournals	Natural Num. excl. 0	upper bound <= 20
maxpermreservebooks	Natural Num. excl. 0	upper bound <= 5
year_joined	Year	current_year-year_joined < 3

Faculty

Name	Type	Constraint on the values
idnumber	UserID	Ref. mapping from global definitions
maxbooks	Natural Num. excl. 0	upper bound<= 40
maxjournals	Natural Num. excl. 0	upper bound<= 50
maxpermreservebooks	Natural Num. excl. 0	upper bound<= 10

A.5 Test Cases for Library Management System

In this section, derivation of test cases is discussed. At each and every state, characterized in the state transition diagrams, test cases are generated for all the transitions that can be fired. The derived test data can then be applied to validate those transitions. Test cases are provided to validate the three operations: *Borrow* , *Reserve* and *Return*. Focusing is done on the most interesting states in the objects that can exercise these transitions. This clearly ensures that transitions are analyzed in complete confidence.

Before generating test cases for the transitions, test cases are provided to validate the *Library* class.

A.5.1 Static Validation of Library

items

T_1 = # items = 0

T_2 = # items = 1

T_3 = # items = upper bound

T_4 = # items < upper bound

borrowers

T_1 = # borrowers = 0

T_2 = # borrowers = 1

T_3 = # borrowers = upper bound

T_4 = # borrowers < upper bound

loans

T_1 = # loans = 0

T_2 = # loans = 1

T_3 = # loans = upper bound

T_4 = # loans < upper bound

reserves

T_1 = # reserves = 0

T_2 = # reserves = 1

T_3 = # reserves = upper bound

T_4 = # reserves < upper bound

It should be noted that the assertions stated as part of the class in the specification must be true at any time.

A.5.2 Test Cases for methods

For better understanding and sake of simplicity this particular operation is split into three operations. The state transition diagrams enclosed provides

a detailed description about this. Test case are derived for each one of them separately.

borrow1: This transition takes place when the status of the LoanItem is *Available.*

borrow2: This is exercised when the status of the LoanItem is *Reserved* and there is no reservation for that item.

borrow3: When the status of the LoanItem is *Reserved* and there are some reservation which are pending, this particular transition is fired.

Test Cases for borrow1

To ensure that the static properties of the input variables are correct, test cases have to be applied to them specifically. Also because of polymorphic substitution, test cases have to be deduced for all possible substitutions.

Parameter 1 : *user*? :↓ *User*

Data items *Ugrad*, *Grad* or *Faculty* can be substituted for an occurrence of *User*. Test cases are generated for each one of them below:

For Ugrad type of user

maxbooks

$T_1 = \{ 1 \}$

$T_2 = \{ 10 \}$

T_3 = { value | 1 > value < 10 }

T_4 = { value | value > 10 }

maxjournals

T_1 = { 1 }

T_2 = { 10 }

T_3 = { value | 1 > value < 10 }

T_4 = { value | value > 10 }

maxpermresbooks

T_1 = { 1 }

T_2 = { 3 }

T_3 = { value | 1 > value < 3 }

T_4 = { value | value > 3 }

current_year - year_joined

T_1 = { 0 }

T_2 = { 3 }

T_3 = { value | 1 > value < 3 }

T_4 = { value | value > 3 }

For Grad type of user

maxbooks

T_1 = { 1 }

T_2 = { 20 }

T_3 = { value | 1 > value < 20 }

$T_4 = \{ \text{value} \mid \text{value} > 20 \}$

maxjournals

$T_1 = \{ 1 \}$

$T_2 = \{ 20 \}$

$T_3 = \{ \text{value} \mid 1 > \text{value} < 20 \}$

$T_4 = \{ \text{value} \mid \text{value} > 20 \}$

maxpermresbooks

$T_1 = \{ 1 \}$

$T_2 = \{ 5 \}$

$T_3 = \{ \text{value} \mid 1 > \text{value} < 5 \}$

$T_4 = \{ \text{value} \mid \text{value} > 5 \}$

current_year-year_joined

$T_1 = \{ 0 \}$

$T_2 = \{ 2 \}$

$T_3 = \{ \text{value} \mid 1 > \text{value} < 2 \}$

$T_4 = \{ \text{value} \mid \text{value} > 2 \}$

For Faculty type of user

maxbooks

$T_1 = \{ 1 \}$

$T_2 = \{ 40 \}$

$T_3 = \{ \text{value} \mid 1 > \text{value} < 40 \}$

$T_4 = \{ \text{value} \mid \text{value} > 40 \}$

maxjournals

$T_1 = \{ 1 \}$

$T_2 = \{ 50 \}$

$T_3 = \{ \text{value} \mid 1 > \text{value} < 50 \}$

$T_4 = \{ \text{value} \mid \text{value} > 50 \}$

maxpermresbooks

$T_1 = \{ 1 \}$

$T_2 = \{ 10 \}$

$T_3 = \{ \text{value} \mid 1 > \text{value} < 10 \}$

$T_4 = \{ \text{value} \mid \text{value} > 10 \}$

For the various test inputs which satisfy the constraints enlisted in the respective mapping the variables must be valid.

Parameter 2 : *item* :↓ *LoanItem*

Test cases are provided for classes, *Book*, *Journal* and *PermReserveBook* which can be substituted for an instance of *LoanItem*.

For Book type of LoanItem

year

$T_1 = \{ \text{value} \mid \text{value} < 1900 \}$

$T_2 = \{ 1900 \}$

$T_3 = \{ 2000 \}$

$T_4 = \{ \text{value} \mid 1900 > \text{value} < 2000 \}$

T_5 = { value | value > 2000 }

status

T_1 = { Available }

T_2 = { Reserved }

T_3 = { Loaned }

T_4 = { Reserved & Loaned }

As other items are dependent on the refinement in either the design and/or implementation, test cases are largely dependent upon those phases.

For Journal type of LoanItem

year

T_1 = { value | value < 1900 }

T_2 = { 1900 }

T_3 = { 2000 }

T_4 = { value | 1900 > value < 2000 }

T_5 = { value | value > 2000 }

volumno

T_1 = { 1 }

T_2 = { 100 }

T_3 = { value | 1 > value < 100 }

T_4 = { value | value > 100 }

issueno

$T_1 = \{ 1 \}$

$T_2 = \{ 24 \}$

$T_3 = \{ \text{value} \mid 1 > \text{value} < 24 \}$

$T_4 = \{ \text{value} \mid \text{value} > 24 \}$

As the other items are dependent upon the refinement in either the design and/or implementation, test cases are largely dependent upon those phases.

For PermReserveBook type of LoanItem

year

$T_1 = \{ \text{value} \mid \text{value} < 1900 \}$

$T_2 = \{ 1900 \}$

$T_3 = \{ 2000 \}$

$T_4 = \{ \text{value} \mid 1900 > \text{value} < 2000 \}$

$T_5 = \{ \text{value} \mid \text{value} > 2000 \}$

loanperiod

$T_1 = \{ 1 \}$

$T_2 = \{ 6 \}$

$T_3 = \{ \text{value} \mid 1 > \text{value} < 6 \}$

$T_4 = \{ \text{value} \mid \text{value} > 6 \}$

As all other items are dependent upon the refinement in either the design and/or implementation test cases are largely dependent upon those phases.

After ensuring that parameters are in conformance to their respective data types and values, operation *borrow1* needs to be validated. In this thesis, we provide test cases only for the valid pre-conditions that are associated with operations. As far as testing is concerned for all possibilities, invalid conditions can be formed simply by negating the pre-conditions in turn.

At each of the interesting states, test cases are derived for the objects. Since, *borrow* operation affects either *book*, *journal* or *permreserve* object, test cases are generated which represent each of them.

State: NotReservedOnShelves

In the *Library* class, the following actions must be valid.

- checkuser - User is present in the list of users
- checkitem - LoanItem is present in the list of items
- checkuserlimit - Borrowed items of user is within limits

a	Transition	:	Borrow_Book		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer,
			status	:	LoanItemStatus,
			authors	:	String,
			publisher	:	String
	Condition	:	status=Available		
	Action	:			
	Input	:			
	Output	:			
	Output	:	Borrow done...		
	Activity	:	Changestatus		
	Input	:	status	:	Available,
			newstatus	:	Loaned
	Output	:	Modified		
			status		
	Events generated	:			

b Transition		:	Borrow_Journal		
Input		:	callno	:	Integer,
			title	:	String,
			year	:	Integer,
			status	:	LoanItemStatus,
			authors	:	String,
			publisher	:	String,
			volumeno	:	Integer,
			issueno	:	Integer
Condition		:	status=Available		
Action		:			
	Input	:			
	Output	:			
Output		:	Borrow Done...		
Activity		:	Changestatus		
	Input	:	status	:	Available,
			newstatus	:	Loaned
	Output	:	Modified		
			status		
Events generated		:			

c	Transition	:	Borrow_PermResBook		
	Input	:	callno	:	Integer
			title	:	String,
			year	:	Integer
			status	:	LoanItemStatus,
			authors	:	String,
			publisher	:	String,
			loanperiod	:	Integer
	Condition	:	status=Available		
	Action	:			
	Input	:			
	Output	:			
	Output	:	Borrow done...		
	Activity	:	Changestatus		
	Input	:	status	:	Available,
			newstatus	:	Loaned
	Output	:	Modified		
			status		
	Events generated	:			

Test Cases for borrow2

As this operation also accepts the same parameters as *borrow1*, test cases for validating the parameters can be taken from the earlier. We provide the table for verifying the dynamic behavior only.

State : ReservedOnshelves

In the *Library* class, the following actions must be valid.

- checkuser - User is present in the list of users
- checkitem - LoanItem is present in the list of items
- checkuserlimit - Borrowed items of user is within limits
- checkreserve - Only this user in reserve queue

a	Transition	:	Borrow_Book		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer,
			status	:	LoanItemStatus,
			authors	:	String,
			publisher	:	String
	Condition	:	status=Reserved		
	Action	:			
	Input	:			
	Output	:			
	Output	:	Borrow done...		
	Activity	:	Changestatus		
	Input	:	status	:	Reserved,
			newstatus	:	Loaned
	Output	:	Modified		
			status		
	Events generated	:			

b	Transition	:	Borrow_Journal		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer,
			status	:	LoanItemStatus,
			authors	:	String,
			publisher	:	String,
			volumeno	:	Integer,
			issueno	:	Integer
	Condition	:	status=Reserved		
	Action	:			
	Input	:			
	Output	:			
	Output	:	Borrow done...		
	Activity	:	Changestatus		
	Input	:	status	:	Reserved,
			newstatus	:	Loaned
	Output	:	Modified		
			status		
	Events generated	:			

Test Cases for borrow3

This operation also accepts the same parameters as *borrow1*. Only the table for verifying the behavior is provided.

State : ReservedOnshelves

In the *Library* class, these actions must be valid.

- checkuser - User is present in the list of users
- checkitem - LoanItem is present in the list of items
- checkuserlimit - Borrowed items of user is within limits
- checkreserve - More then this user in the reservation queue

a	Transition	:	Borrow_Book		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer,
			status	:	LoanItemStatus,
			authors	:	String,
			publisher	:	String
	Condition	:	status=Reserved		
	Action	:			
	Input	:			
	Output	:			
	Output	:	Borrow Done...		
	Activity	:	Changestatus		
	Input	:	status	:	Reserved,
			newstatus	:	Reserved & Loaned
	Output	:	Modified		
			status		
	Events generated	:			

b	Transition	:	Borrow_Journal		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer,
			status	:	LoanItemStatus,
			authors	:	String,
			publisher	:	String,
			volumeno	:	Integer,
			issueno	:	Integer
	Condition	:	status=Reserved		
	Action	:			
	Input	:			
	Output	:			
	Output	:	Borrow Done...		
	Activity	:	Changestatus		
	Input	:	status	:	Reserved,
			newstatus	:	Reserved & Loaned
	Output	:	Modified		
			status		
	Events generated	:			

State : ItemLoaned

This operation takes in only one parameter, *LoanItem*. And hence static validation need to be done for LoanItem and its substitutable objects only.

In the *Library* class, these actions must be valid.

- checkitem - LoanItem is present in the list of items

a	Transition	:	Return_Book		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer,
			status	:	LoanItemStatus,
			authors	:	String,
			publisher	:	String
	Condition	:	status=Loaned		
	Action	:			
	Input	:			
	Output	:			
	Output	:	Return done...		
	Activity	:	Changestatus		
	Input	:	status	:	Loaned,
			newstatus	:	Available
	Output	:	Modified		
			status		
	Events generated	:			

b	Transition	:	Return_Journal		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer,
			status	:	LoanItemStatus,
			authors	:	String,
			publisher	:	String,
			volumeno	:	Integer,
			issueno	:	Integer
	Condition	:	status=Loaned		
	Action	:			
	Input	:			
	Output	:			
	Output	:	Return done...		
	Activity	:	Changestatus		
	Input	:	status	:	Loaned,
			newstatus	:	Available
	Output	:	Modified		
			status		
	Events generated	:			

c	Transition	:	Return_PermReserveBook		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer ,
			status	:	LoanItemStatus ,
			authors	:	String,
			publisher	:	String,
			loanperiod	:	Integer
	Condition	:	status=Loaned		
	Action	:			
		Input :			
		Output :			
	Output	:	Return done...		
	Activity	:	Changestatus		
		Input :	status	:	Loaned,
			newstatus	:	Available
		Output :	Modified		
			status		
	Events				
	generated	:			

State : ItemLoaned

The operation, *reservation* can be exercised only for these valid conditions:

- checkitem - LoanItem must not be a Permanent Reserved Book
- checkjouuser - If LoanItem is a Journal then it can't be reserved by an Undergraduate

a	Transition	:	Reserve_Book		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer ,
			status	:	LoanItemStatus ,
			authors	:	String,
			publisher	:	String
	Condition	:	status ! = Available		
	Action	:			
	Input	:			
	Output	:			
	Output	:	Reserv. done...		
	Activity	:	Changestatus		
	Input	:	status	:	Loaned,
			newstatus	:	Reserved & Loaned
	Output	:	Modified status		
	Events generated	:			

b	Transition	:	Reserve_Journal		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer ,
			status	:	LoanItemStatus ,
			authors	:	String,
			publisher	:	String,
			volumeno	:	Integer,
			issueno	:	Integer.
	Condition	:	status ! = Available,		
			user ! = Ugrad		
	Action	:	checkvaliduser		
	Input	:	user	:	User [polymorphic],
			Collection of user		
	Output	:	Boolean;		Undergrad or not
	Action	:	checkvaliditem		
	Input	:	item	:	LoanItem [polymorphic],
			Collection of item		
	Output	:	Boolean;		Unique or Not unique
	Output	:	Reserv. done...		
	Activity	:	Changestatus		
	Input	:	status	:	Loaned,
			newstatus	:	Reserved & Loaned
	Output	:	Modified status		
	Events generated	:			

State : ReservedItemLoaned

Parameter for this operation, *LoanItem* is validated taking the test cases from *borrow1*. In the *Library* class, following action must be valid.

- checkitem - LoanItem is present in the list of items

a	Transition	:	Return_Book		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer ,
			status	:	LoanItemStatus,
			authors	:	String,
			publisher	:	String
	Condition	:	status= Reserved & Loaned		
	Action	:			
	Input	:			
	Output	:			
	Output	:	Return done...		
	Activity	:	Changestatus		
	Input	:	status	:	Reserved & Loaned,
			newstatus	:	Reserved
	Output	:	Modified status		
	Events generated	:			

b	Transition	:	Return_Journal		
	Input	:	callno	:	Integer,
			title	:	String,
			year	:	Integer ,
			status	:	LoanItemStatus ,
			authors	:	String,
			publisher	:	String,
			volumeno	:	Integer ,
			issueno	:	Integer
	Condition	:	status=		
			Reserved & Loaned		
	Action	:			
	Input	:			
	Output	:			
	Output	:	Return done...		
	Activity	:	Changestatus		
	Input	:	status	:	Reserved & Loaned.
			newstatus	:	Reserved
	Output	:	Modified		
			status		
	Events generated	:			

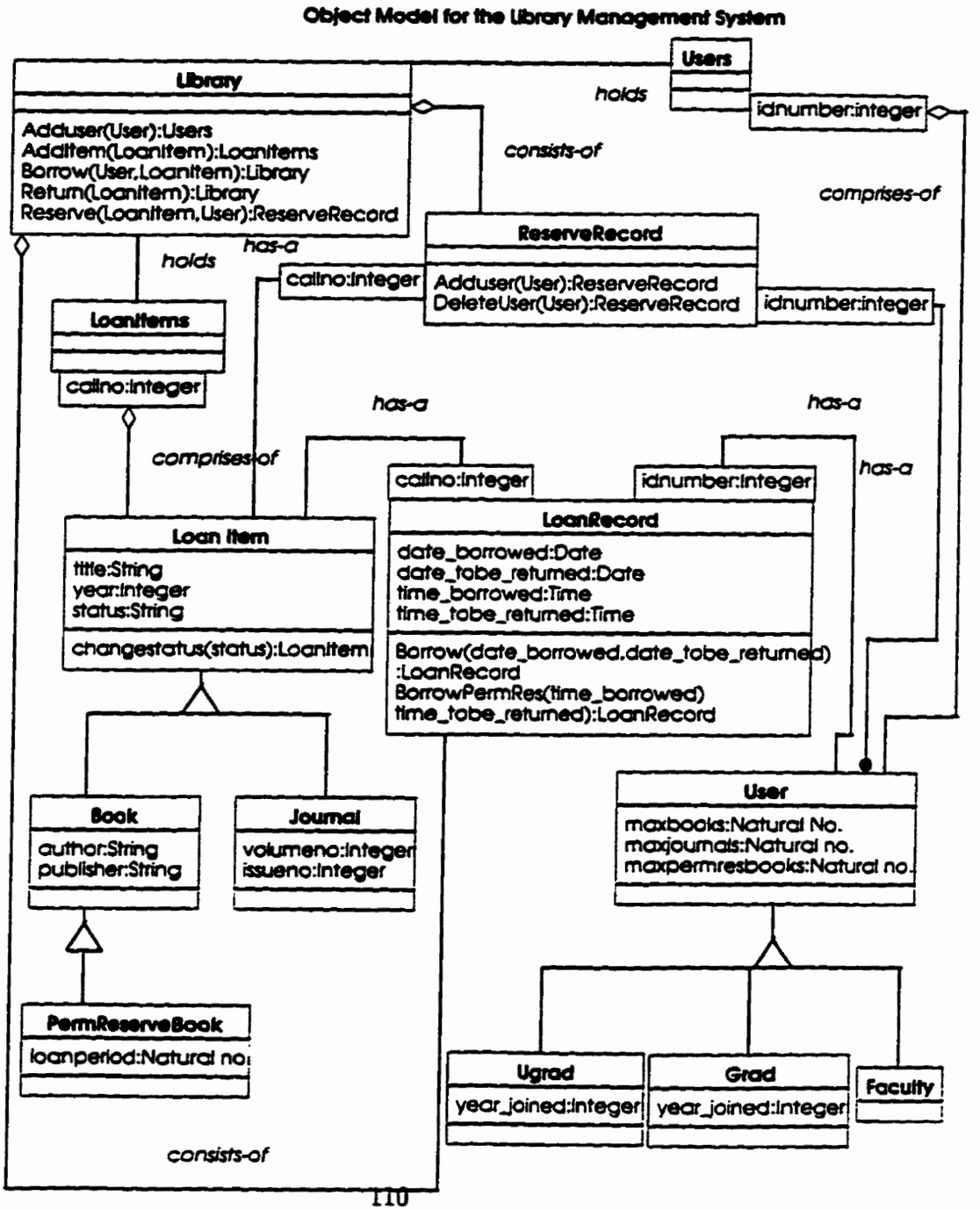


Figure A.1: Object Model for Library Management System

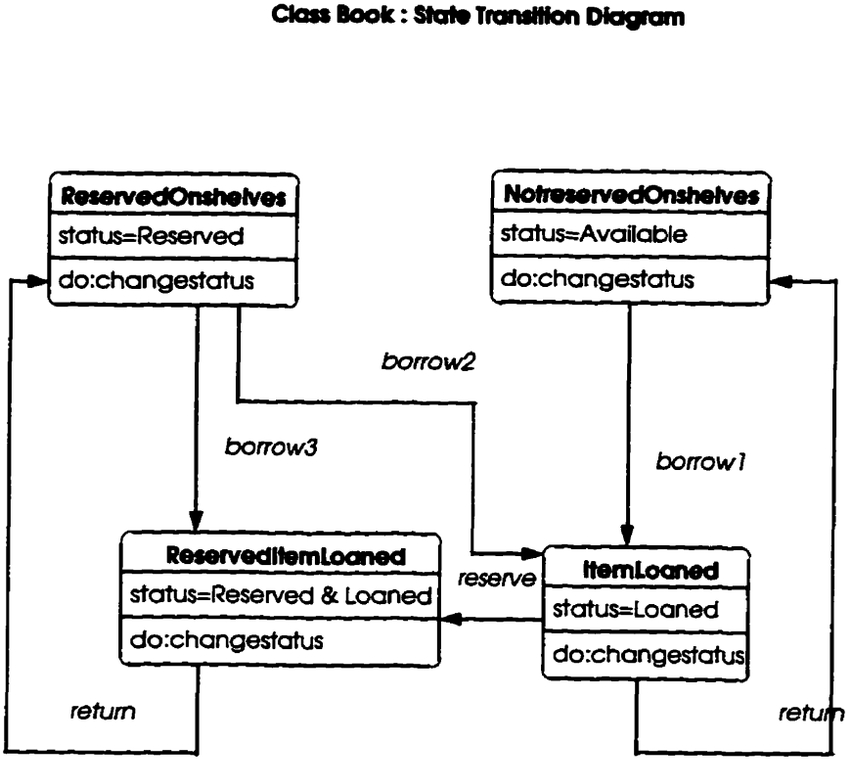


Figure A.2: State Transition Diagram for *Book*

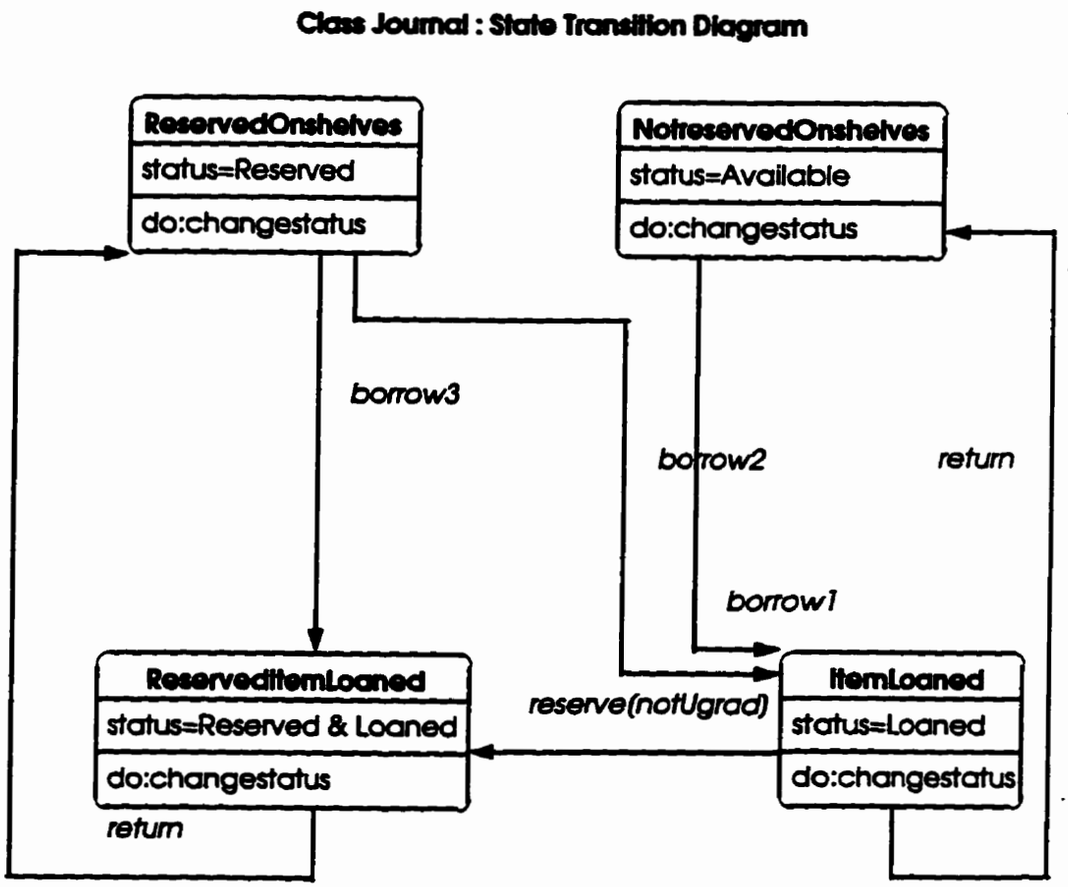


Figure A.3: State Transition Diagram for *Journal*

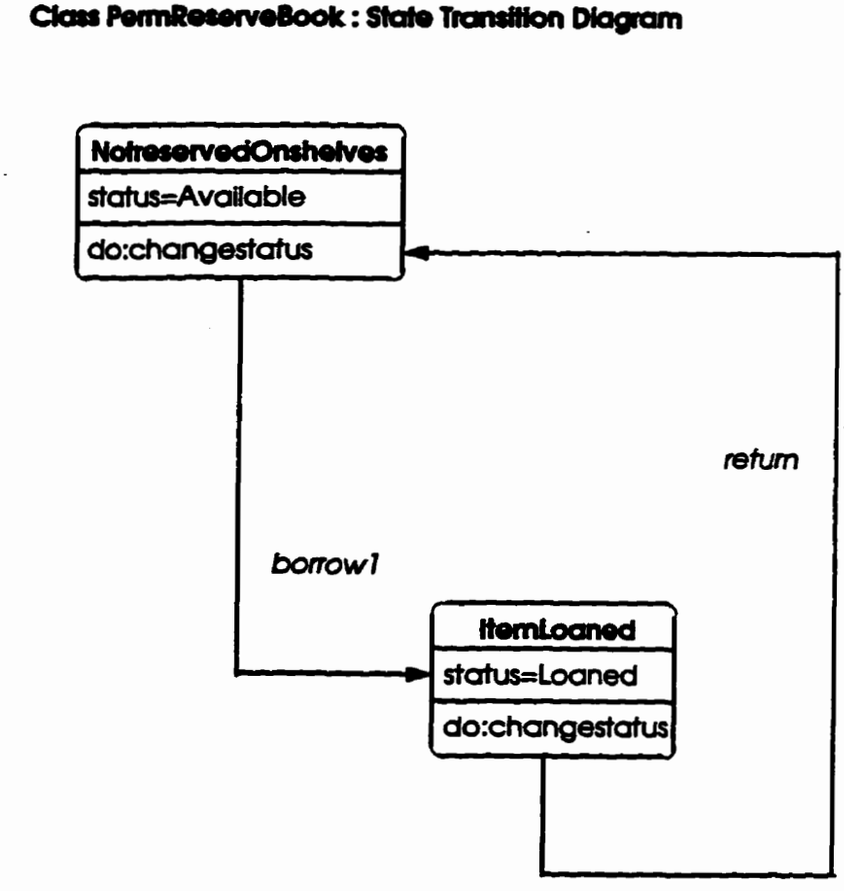


Figure A.4: State Transition Diagram for *PermReserveBook*

Appendix B

Automated Teller Machine

B.1 Problem Description

In this section, an object-oriented specification of an *automated teller machine (ATM)* in a bank environment is presented. The specification is written using Object-Z notation. Current version of this specification has not been type-checked.

An ATM permits a user to invoke a finite set of transactions from the machine. The basic requirements are that a user must have a valid identification and at least one account with the bank. Users are given identification cards which are validated before any transaction begins. Details of assumptions and requirements will be described subsequently. Any malfunctions such as card being not returned to the user, card being damaged or unexpected mechanical failure of the machine are not taken into account.

The following are the assumptions and requirements for the automated teller machine being specified:

1. Each user of the machine has an identification card which carries a unique number. This number uniquely identifies a user and consequently a set of accounts owned by this user. Within the bank records, this number is considered to be the user's identification number. Consequently, two users cannot share a card. For the purposes of this specification, the card is not modeled. Consequently, there will not be operations to insert, remove or other operations on cards.

2. A user can have more than one account.

3. An account is specified by its unique account number, unique type, the owner of the account and the current balance. Depending on the type of the account, it may have further attributes such as interest rate, ending date for a term deposit and so on. Moreover, the type of transactions that can be exercised on an account also depend on the type of the account. For example, no withdrawal is permitted for a fixed-deposit account.

4. The following two transactions are permitted on all accounts - *deposit* and *check balance.*

5. There are three types of users - *General, Privileged* and *Staff. General* refers to public, *Privileged* refers to a set of users who must have at least one account in each account type (described below). Further, a

Privileged user can withdraw money from a fixed-deposit account under certain conditions (described below). *Staff* refers to the staff member of the bank who have authorization to invoke any valid transaction. There is at least one transaction (updating the balance of the money pool in the machine) which can only be invoked by a staff of the bank and by no user from *Privileged* or *General.*

6. There are three types of accounts - *Savings, Chequing* and *Fixed-Deposit.*

7. A savings account is the simplest type of account in the bank. *Withdrawal* is permitted in savings account. Interest is calculated on a daily basis for a savings account and the interest rate is fixed and known to the bank initially.

8. A Chequing account is similar to savings account but there should be a minimum balance maintained at all times. Consequently, *withdrawal* is restricted. Once again, interest is calculated as in savings account for a different, but fixed, interest rate which is known to the bank initially.

9. The two transactions that can be invoked on a fixed-deposit account are *deposit* and *close-withdraw.* A fixed-deposit account has a starting date and an ending date. Any amount deposited before the ending date is added to the existing balance and interest is calculated on a daily basis for a fixed interest rate (once again known to the bank initially). At the end of the term, the account is closed and the money is withdrawn.

10. A fixed-deposit account has a minimum balance with which it could be opened initially. The owner of a fixed-deposit account can deposit

additional money before the term expires. This user can also withdraw 10% of the current balance at any time before the ending date if the following conditions are satisfied:

- The user must be a *Privileged* user.
- The current balance of the fixed-deposit account must be at least 110% of the minimum balance for this account (and hence after the withdrawal, the account still maintains its minimum balance).

11. All accounts are assumed to exist initially since a user cannot open a new account with the machine (in a real-environment, this is done by the bank staff). All accounts are assumed to be opened with an initial balance greater than zero.

12. There is a money pool in the machine which is initially loaded with a fixed amount. Every withdrawal consequently reduces this amount. If there is no money in the pool, no further withdrawal is permitted; however, deposits and checking of balances could be done. This money pool will be updated only by a staff member in the bank who also uses an identification card.

13. Depositing a cheque is considered to be equivalent to depositing cash, except that the money pool is not updated.

B.2 Global Definitions

Following are the global definitions shared by all class definitions given in the next section.

$[IDNUMBER, ACCOUNT\_NUMBER]$

B.3 Class Definitions

We now describe the individual class definitions.

User

$$id : IDNUMBER$$
$$accounts : \mathbb{P}\, ACCOUNT\_NUMBER$$

INIT

$$accounts = \varnothing$$

AddAccount

$$\Delta(accounts)$$
$$new\_account? : ACCOUNT\_NUMBER$$

$$accounts' = accounts \cup \{new\_account?\}$$

DeleteAccount

$$\Delta(accounts)$$
$$old\_account? : ACCOUNT\_NUMBER$$

$$old\_account? \in accounts$$
$$accounts' = accounts \setminus \{old\_account?\}$$

A user has a unique i.d. number and a set of accounts. These accounts are represented by the account numbers in a user record since the definition

of account has not been given yet. There are two operations in the class *User* which permit one to add and delete an account to the accounts owned by the user. By default, all the attributes and operations of the class *User* are visible to other classes.

Account

$number : ACCOUNT\_NUMBER$
$balance : \mathbb{R}$
$interest\_rate : \mathbb{R}$
$interest\_calculated : Date$
$owner : IDNUMBER$

$balance \geq 0$

INIT

$balance \geq 0$

Deposit

$\Delta(balance)$
$amount? : \mathbb{R}$

$balance' = balance + amount?$

CheckBalance

$amount! : \mathbb{R}$

$amount! = balance$

Calculate_Interest

$\Delta(balance)$

$current\_date > interest\_calculated$
$balance' = balance + balance *$
$((current\_date - interest\_calculated)/365) * interest\_rate$

The class *Account* denotes an abstract account which specifies only the essential characteristics of all the accounts such as the account number, owner,

balance and interest rate. Notice that interest rate may vary with respect to each account and each type of account. It has an additional field which denotes the last date at which interest was calculated. The three transactions permitted on this account (in fact on any account) are *Deposit CheckBalance* and *Calculate_Interest*. The last of these transactions calculates interest for the period from the last date of interest calculation (indicated by the variable *interest_calculated*) up to the current date. The term *current_date* in this operation is assumed to be defined later in the design/implementation which returns the current date. As in *User*, all attributes and operations of *Account* are visible.

Savings

Account

Withdraw

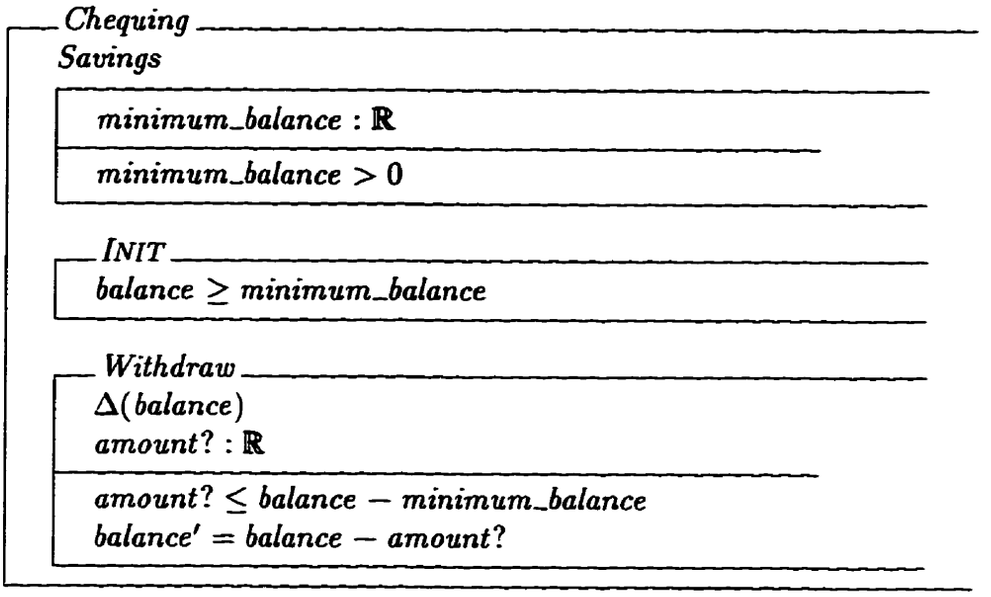
$\Delta(balance)$

$amount? : \mathbb{R}$

$amount? \leq balance$

$balance' = balance - amount?$

A savings account has an additional transaction *Withdraw*.



A Chequing account is similar to a savings account except for maintaining the minimum balance and restricting withdrawal accordingly.

FixedDeposit

Account

$minimum\_balance : \mathbb{R}$

$start, end : Date$

$minimum\_balance > 0$

$end > start$

INIT

$balance \geq minimum\_balance$

Withdraw

$\Delta(balance)$

$amount? : \mathbb{R}$

$amount? \leq 0.1 * balance$

$balance - amount? \geq minimum\_balance$

$balance' = balance - amount?$

CloseWithdraw

$return! : \mathbb{R}$

$end = current\_date$

$balance' = 0$

$return! = balance$

Calculate_Interest

$interest\_calculated \geq start$

$current\_date \leq end$

A fixed deposit account has a minimum balance to be maintained throughout its term period (denoted by *start* and *end*). Initially, the account is opened with its balance at least equal to this minimum balance. There are two operations in this class: *Withdraw* which will only be invoked by a privileged user (constrained later at the bank level) and *CloseWithdraw* which will be

invoked at the time of closing this account. The operation *Calculate_Interest* introduces additional constraints to the same operation defined in *Account* to ensure that the interest is calculated for the period of the Fixed Deposit.

General

User

Privileged

User

$$\exists\, a_1, a_2, a_3 :\downarrow \mathit{Account} \mid$$
$$a_1.\mathit{number} \in \mathit{accounts} \land a_2.\mathit{number} \in \mathit{accounts} \land$$
$$a_3.\mathit{number} \in \mathit{accounts} \bullet$$
$$a_1 \in \mathit{Savings} \land a_2 \in \mathit{Chequing} \land a_3 \in \mathit{FixedDeposit}$$

Staff

Privileged

The classes *General* and *Staff* seem to be redundant at this stage since there is only very little information provided in the problem description about the distinction between the users. Further, users are distinguished by the authorizations to invoke certain transactions. We assume that further details for these classes will be provided during design and implementation which make them structurally distinct.

The class *Privileged* is structurally distinguished from other users by possessing at least three accounts, one in each type.

We now define the bank.

Bank

$$users : \mathbb{P} \downarrow User$$
$$accounts : \mathbb{P} \downarrow Account$$
$$money\_pool : \mathbb{R}$$

$$money\_pool \geq 0$$
$$\bigcup \{u : User \mid u \in users \bullet u.accounts\} = \{a : Account \mid a \in accounts \bullet a.number\}$$
$$\{a : Account \mid a \in accounts \bullet a.owner\} = \{u : User \mid u \in users \bullet u.id\}$$
$$\forall u : users \bullet (u \in General \lor u \in Privileged \lor u \in Staff)$$
$$\forall a : accounts \bullet a \in Savings \lor a \in Chequing \lor a \in FixedDeposit$$

INIT

$$users = \varnothing$$
$$accounts = \varnothing$$
$$money\_pool > 0$$

OpenAccount$_0$

$$\Delta(users, accounts)$$
$$new? :\downarrow Account$$
$$user? :\downarrow User$$

$$user? \in users$$
$$new?.number \notin \{a :\downarrow Account \mid a \in accounts\}$$
$$accounts' = accounts \cup \{new?\}$$

$$OpenAccount \mathrel{\widehat=} OpenAccount_0 \bullet user?.AddAccount[new?.number/new\_account?]$$

Bank(Contd...)

CloseAccount$_0$

$\Delta(users, accounts)$
$old? :\downarrow Account$
$user? :\downarrow User$

$user? \in users$
$old? \in accounts$
$old?.number \in user.accounts$
$old?.balance = 0$
$accounts' = accounts \setminus \{old?\}$

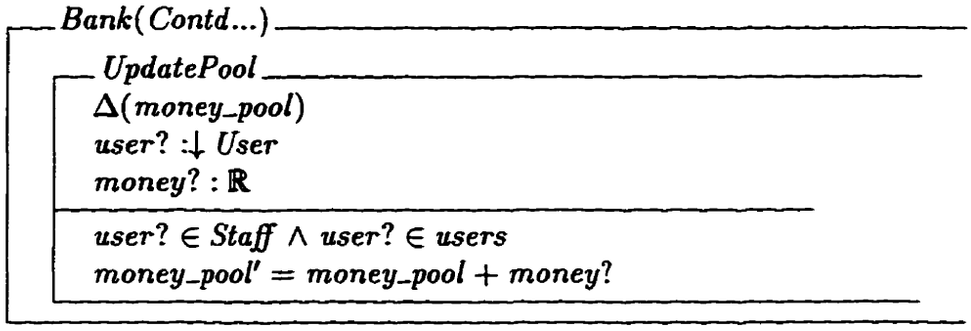
$CloseAccount \mathrel{\widehat=} CloseAccount_0 \bullet$
$user?.DeleteAccount[old?.number / old\_account?] \land$
$[old? \in FixedDeposit] \Rightarrow old?.CloseWithdraw$

SelectUserAccount

$user? :\downarrow User$
$account? :\downarrow Account$

$user? \in users$
$account? \in accounts$
$account?.number \in user?.accounts$

$Deposit \mathrel{\widehat=} SelectUserAccount \bullet account?.Deposit \bullet$
$[money\_ppol' = money\_pool + amount?]$
$CheckBalance \mathrel{\widehat=} SelectUserAccount \bullet account?.CheckBalance$
$Calculate\_Interest \mathrel{\widehat=} SelectUserAccount \bullet account?.Calculate\_Interest$
$Withdraw \mathrel{\widehat=} SelectUserAccount \bullet (account?.Withdraw \land$
$account? \in FixedDeposit \Rightarrow user? \in Privileged) \bullet$
$[money\_pool' = money\_pool - amount?]$



The class *Bank* contains three attributes: *user* defining the set of users/clients of the bank, *accounts* defining the set of accounts created and manipulated in the bank and *money_pool* which denotes the money available in the bank. The first two attributes are defined to be polymorphic and hence users of all the three types and accounts of all the three types are included. Further, we constrain that instances of the general classes *User* and *Account* should no longer be used; this is indicated by the constraints that (i) each user must belong to one of the three categories *General, Staff* and *Privileged*; similarly (ii) each account must be one of the three types *Savings, Chequing* and *FixedDeposit.* The operations *OpenAccount* and *CloseAccount* define the opening and closing of new and existing account respectively. Notice that each one of these operations is defined in two parts, (i) validating the input parameters and (ii) invoking the appropriate operation in the class *User.* This is necessary because an operation, when defined in box notation cannot invoke another operation. In addition, if the account being closed is a fixed deposit account, then the money should be withdrawn before closing the account.

The operation *SelectUserAccount* selects a user and an account for further operations and validates them. As seen from the specification, all the rest

of the operations in this class use this operation. The rest of the operations are defined promoting the lower level operations in the individual account class. Notice that both *Deposit* and *Withdraw* affect the money pool. In case of *Withdraw*, we also ensure that the user must be a privileged user if the account is a fixed deposit account. The last operation *UpdatePool* is defined to update the money pool in the bank. As asserted by the constraints, this operation can only be invoked by a staff of the bank.

B.4 Mapping of Types

In this section, a mapping is provided for the types in the specification to their corresponding implementation data types.

B.4.1 Mapping of Global Definitions

Type	Mapped Type	Constraint on the values
IDNUMBER	User Defined	Ref. implementation or design
ACCOUNT_NUMBER	User Defined	Ref. implementation or design

B.4.2 Mapping of Class Definitions

User

Name	Type	Constraint on the values
id	IDNUMBER	Ref. mapping from global definitions
accounts	$\mathbb{P}$ ACCOUNT_NUMBER	Ref. mapping of global definitions

Account

Name	Type	Constraint on the values
number	ACCOUNT_NUMBER	Ref. global definition mapping
balance	Real	Ref. design or implementation
interest_rate	Real	Ref. design or implementation
interest_calculated	Date	Ref. design or implementation
owner	IDNUMBER	Ref. global definitions mapping

Savings

Name	Type	Constraint on the values
number	ACCOUNT_NUMBER	Ref. global definition mapping
balance	Real	Ref. design or implementation
interest_rate	Real	Ref. design or implementation
interest_calculated	Date	Ref. design or implementation
owner	IDNUMBER	Ref. global mapping

Chequing

Name	Type	Constraint on the values
number	ACCOUNT_NUMBER	Ref. global mapping
balance	Real	Ref. design or implementation
interest_rate	Real	Ref. design or implementation
interest_calculated	Date	Ref. design or implementation
owner	IDNUMBER	Ref. global mapping
minimum_balance	Real	Ref. design or implementation

Fixed_Deposit

Name	Type	Constraint on the values
number	ACCOUNT_NUMBER	Ref. global mapping
balance	Real	Ref. design or implementation
interest_rate	Real	Ref. design or implementation
interest_calculated	Date	Ref. design or implementation
owner	IDNUMBER	Ref. global mapping
minimum_balance	Real	Ref. design or implementation
start	Date	Ref. design or implementation
end	Date	Ref. design or implementation

As classes *General*, *Privileged* and *Staff* do not add any new attributes, test cases for the static validation *User* holds good.

Bank

Name	Type	Constraint on the values
money_pool	Real	Ref. design or implementation for information.

B.5 Test cases for Automated Teller Machine

From the dynamic model of this system, operations *OpenAccount*, *CloseAccount* are chosen to be focussed in detail. Test cases are derived at the *Bank* level. Interesting states are identified in the corresponding state transition diagrams and test cases are generated for all the transitions that can be fired. Also, synchronization of events by transitions are identified. This will fire transitions in other objects and test cases are also generated for them in their corresponding object classes.

B.5.1 Static Validation of Bank

users

T_1 = # users = 0

T_2 = # users = 1

T_3 = # users = upper bound

T_4 = # users < upper bound

accounts

T_1 = # accounts = 0

T_2 = # accounts = 1

T_3 = # accounts = upper bound

T_4 = # accounts < upper bound

money_pool

T_1 = { value | value < lowerbound }

T_2 = { lowerbound }

T_3 = { 0 }

T_4 = { value | upperbound < value > lowerbound }

T_5 = { upperbound }

T_6 = { value | value > upperbound }

where the upperbound and lowerbound values are taken from either design or implementation.

It should be clearly noted that assertions which are stated as part of this class in specification must be valid at any given point of time.

B.5.2 Test cases for methods

Initially, validation of the static entities present in this operation must be performed. As this case study also involves polymorphic substitutions of ob-

jects, test cases are deduced in such a manner that it takes care of all possible scenarios.

Parameter 1 : *user*? :↓ *User*

As the inherited classes of *User*, which are polymorphically substitutable. do not add any new attributes, test cases for static validation is provided only *User*'s attributes.

id

T_1 = { lowerbound value }

T_2 = { 0 }

T_3 = { value | lowerbound > value < upperbound }

T_4 = { upperbound }

T_5 = { value |> upperbound }

accounts

T_1 = # accounts = 0

T_2 = # accounts = 1

T_3 = # accounts = upper bound

T_4 = # accounts < upper bound

Parameter 2 : *accounts* :↓ *Account*

For Savings type of account

This is simply, an *Account* class without any addition of properties.

balance

T_1 = { 0 }

T_2 = { upper bound }

T_3 = { value | 0 > value < upper bound }

T_4 = { value | value > upper bound }

interest_rate

T_1 = { 0 }

T_2 = { upper bound }

T_3 = { value | 0 > value < upper bound }

T_4 = { value | value > upper bound }

interest_calculated

Test suites follow design and/or implementation strategies for implementation of date.

owner

T_1 = { value | value < lower bound }

T_2 = { lower bound }

T_3 = { value | lower bound > value < upper bound }

T_4 = { upper bound }

T_5 = { value | value > upper bound }

For Chequing type of account

This is simply, an *Account* class without any addition of properties. The test case for this,

balance

T_1 = { 0 }

T_2 = { upper bound }

T_3 = { value | 0 > value < upper bound }

T_4 = { value | value > upper bound }

interest_rate

T_1 = { 0 }

T_2 = { upper bound }

T_3 = { value | 0 > value < upper bound }

T_4 = { value | value > upper bound }

interest_calculated

The test set for this follow constraints that have been stated in design and/or implementation for Date data type.

owner

T_1 = { value | lower bound < value }

T_2 = { lower bound }

T_3 = { value | lower bound > value < upper bound }

T_4 = { upper bound }

T_5 = { value | value > upper bound }

minimum_balance

T_1 = { 0 }

T_2 = { upper bound }

T_3 = { value | 0 > value < upperbound }

For Fixed_Deposit type of account

balance

T_1 = { 0 }

T_2 = { upper bound }

T_3 = { value | 0 > value < upper bound }

T_4 = { value | value > upper bound }

interest_rate

T_1 = { 0 }

T_2 = { upper bound }

T_3 = { value | 0 > value < upper bound }

T_4 = { value | value > upper bound }

interest_calculated

Depends upon constraint in design / implementation

owner

T_1 = { value | value < lower bound }

T_2 = { lower bound }

T_3 = { value | lower bound > value < upper bound }

T_4 = { upper bound }

T_5 = { value | value > upper bound }

minimum_balance

T_1 = { 0 }

T_2 = { upperbound }

T_3 = { value | 0 > value < upper bound }

start

Test cases depends upon constraint specified in design and or implementation.

end

Test cases depends upon constraint in design/implementation.

State : Initial State

In the *Bank* class, these actions must be valid.,

- checkuser - User is present in the list of users
- checknumber - number must not be present in the bank already

a	Transition	:	OpenAccount_0		(Savings)
	Input	:	number	:	ACCOUNT_NUMBER,
			balance	:	Real,
			interest_rate	:	Real,
			interest_calculated	:	Date,
			owner	:	IDNUMBER
	Conditions	:			
	Action	:	checkvaliduser		
	Input	:	user	:	user [polymorphic],
			Collection of users		
	Output	:	Boolean;		Unique or Not unique
	Action	:	checkaccount		
	Input	:	new	:	Account [polymorphic],
			accounts collection		
	Output	:	Boolean;		Unique or Not unique
	Output	:	Account Opened...		
	Activity	:	AddAccount		
	Input	:	new	:	Account [polymorphic],
			accounts	:	$\mathbb{P}$ Account [polymorphic]
	Output	:	Acc. added		
	Events generated	:	Addaccount in User		;

b	Transition	:	OpenAccount_0		(Chequing)
	Input	:	number	:	ACCOUNT_NUMBER,
			balance	:	Real,
			interest_rate	:	Real,
			interest_calculated	:	Date,
			owner	:	IDNUMBER,
			minimum_balance	:	Real
	Conditions	:			
	Action	:	checkvaliduser		
	Input	:	user	:	user [polymorphic],
			Collection of users		
	Output	:	Boolean;		Unique or Not unique
	Action	:	checkaccount		
	Input	:	new	:	Account [polymorphic],
			accounts collection		
	Output	:	Boolean;		Unique or Not unique
	Output	:	Account Opened...		
	Activity	:	AddAccount		
	Input	:	new	:	Account [polymorphic],
			accounts	:	$\mathbb{P}$ Account [polymorphic]
	Output	:	Acc. added		
	Events generated	:	Addaccount in User		;

c	Transition	:	OpenAccount_0		(Fixed_Deposit)
	Input	:	number	:	ACCOUNT_NUMBER,
			balance	:	Real,
			interest_rate	:	Real,
			interest_calculated	:	Date,
			owner	:	IDNUMBER,
			minimum_balance	:	Real,
			start	:	Date,
			end	:	Date
	Conditions	:			
	Action	:	checkvaliduser		
	Input	:	user	:	user [polymorphic],
			Collection of users		
	Output	:	Boolean;		Unique or Not unique
	Action	:	checkaccount		
	Input	:	new	:	Account [polymorphic],
			Accounts collection		
	Output	:	Boolean;		Unique or Not unique
	Output	:	Account Opened...		
	Activity	:	AddAccount		
	Input	:	new	:	Account [polymorphic],
			accounts	:	$\mathbb{P}$ Account [polymorphic]
	Output	:	Acc. added		
	Events	:	Addaccount		;
	generated		in User		

At Class User level

Transition		:	AddAccount	
Input		:	new_account	: ACCOUNT_NUMBER,
			accounts	: ℙ ACCOUNT_NUMBER
Conditions		:		
Action		:		
	Input	:		
	Output	:		
Output		:	Account Added.....	
Activity		:	addaccount	
	Input	:	new_account	: ACCOUNT_NUMBER.
			accounts	: ℙ Account [polymorphic]
	Output	:	Acc. added	
Events generated		:		

State : Account Opened

The following actions must be valid at the Bank level.,

- checkuser - User must be already present in the users
- checkaccount - Account must be in the bank already
- checknumber - Acc. number must be present in the user
- checkbalance - Balance of the account must be zero

a	Transition	:	CloseAccount_0		(Savings)
	Input	:	number	:	ACCOUNT_NUMBER,
			balance	:	Real,
			interest_rate	:	Real,
			interest_calculated	:	Date,
			owner	:	IDNUMBER
	Conditions	:			
	Action	:	checkvaliduser		
	Input	:	user	:	user [polymorphic],
			Collection of users		
	Output	:	Boolean;		Unique or Not unique
	Action	:	checkaccount		
	Input	:	new	:	Account [polymorphic],
			Collection of accounts		
	Output	:	Boolean;		Unique or Not unique
	Action	:	checkuseraccount		
	Input	:	new.number	:	ACCOUNT_NUMBER,
			user	:	User [polymorphic]
	Output	:	Boolean;		Present or Not present
	Action	:	checkbalance		
	Input	:	balance	:	Real
	Output	:	Boolean;		0 or not

contd... on next page

contd... from previous page					
Output		:	Account Deleted...		
Activity		:	CloseAccount		
	Input	:	new	:	Account [polymorphic],
			accounts	:	$\mathbb{P}$ Account [polymorphic]
	Output	:	Acc. deleted		
Events generated		:	DeleteAccount in User		;

b	Transition	:	CloseAccount_0		(Chequing)
	Input	:	number	:	ACCOUNT_NUMBER,
			balance	:	Real,
			interest_rate	:	Real,
			interest_calculated	:	Date,
			owner	:	IDNUMBER,
			minimum_balance	:	Real
	Conditions	:			
	Action	:	checkvaliduser		
	Input	:	user	:	user [polymorphic],
			Collection of users		
	Output	:	Boolean;		Unique or Not unique
	Action	:	checkaccount		
	Input	:	new	:	Account [polymorphic],
			Collection of accounts		
	Output	:	Boolean;		Unique or Not unique
	Action	:	checkuseraccount		
	Input	:	new.number	:	ACCOUNT_NUMBER.
			user	:	User [polymorphic]
	Output	:	Boolean;		Present or Not present
	Action	:	checkbalance		
	Input	:	balance	:	Real

contd... on next page

contd... from previous page					
	Output	:	Boolean;		0 or not
Output		:	Account Deleted...		
Activity		:	CloseAccount		
	Input	:	new	:	Account [polymorphic],
			accounts	:	$\mathbb{P}$ Account [polymorphic]
	Output	:	Acc. deleted		
Events generated		:	DeleteAccount in User		;

c	Transition	:	CloseAccount_0		(Fixed Deposit)
	Input	:	number	:	ACCOUNT_NUMBER,
			balance	:	Real,
			interest_rate	:	Real,
			interest_calculated	:	Date,
			owner	:	IDNUMBER,
			minimum_balance	:	Real,
			start	:	Date,
			end	:	Date
	Conditions	:			
	Action	:	checkvaliduser		
	Input	:	user	:	user [polymorphic],
			Collection of users		
	Output	:	Boolean;		Unique or Not unique
	Action	:	checkaccount		
	Input	:	new	:	Account [polymorphic],
			Collection of accounts		
	Output	:	Boolean;		Unique or Not unique
	Action	:	checkuseraccount		
	Input	:	new.number	:	ACCOUNT_NUMBER,
			user	:	User [polymorphic]
	Output	:	Boolean;		Present or Not present

contd... on next page

contd... from previous page					
Action		:	checkbalance		
	Input	:	balance	:	Real
	Output	:	Boolean;		0 or not
Output		:	Account Deleted...		
Activity		:	CloseAccount		
	Input	:	new	:	Account [polymorphic],
			accounts	:	$\mathbb{P}$ Account
					[polymorphic]
	Output	:	Acc. deleted		
Events		:	DeleteAccount		;
generated			in User		

At Class User level

Transition		:	DeleteAccount		
Input		:	old_account	:	ACCOUNT_NUMBER,
			accounts	:	ℙ ACCOUNT_NUMBER
Conditions		:			
Action		:			
	Input	:			
	Output	:			
Output		:	Account Deleted.....		
Activity		:	delaccount		
	Input	:	old_account	:	Account [polymorphic],
			accounts	:	ℙ Account [polymorphic]
	Output	:	Acc. removed		
Events generated		:			

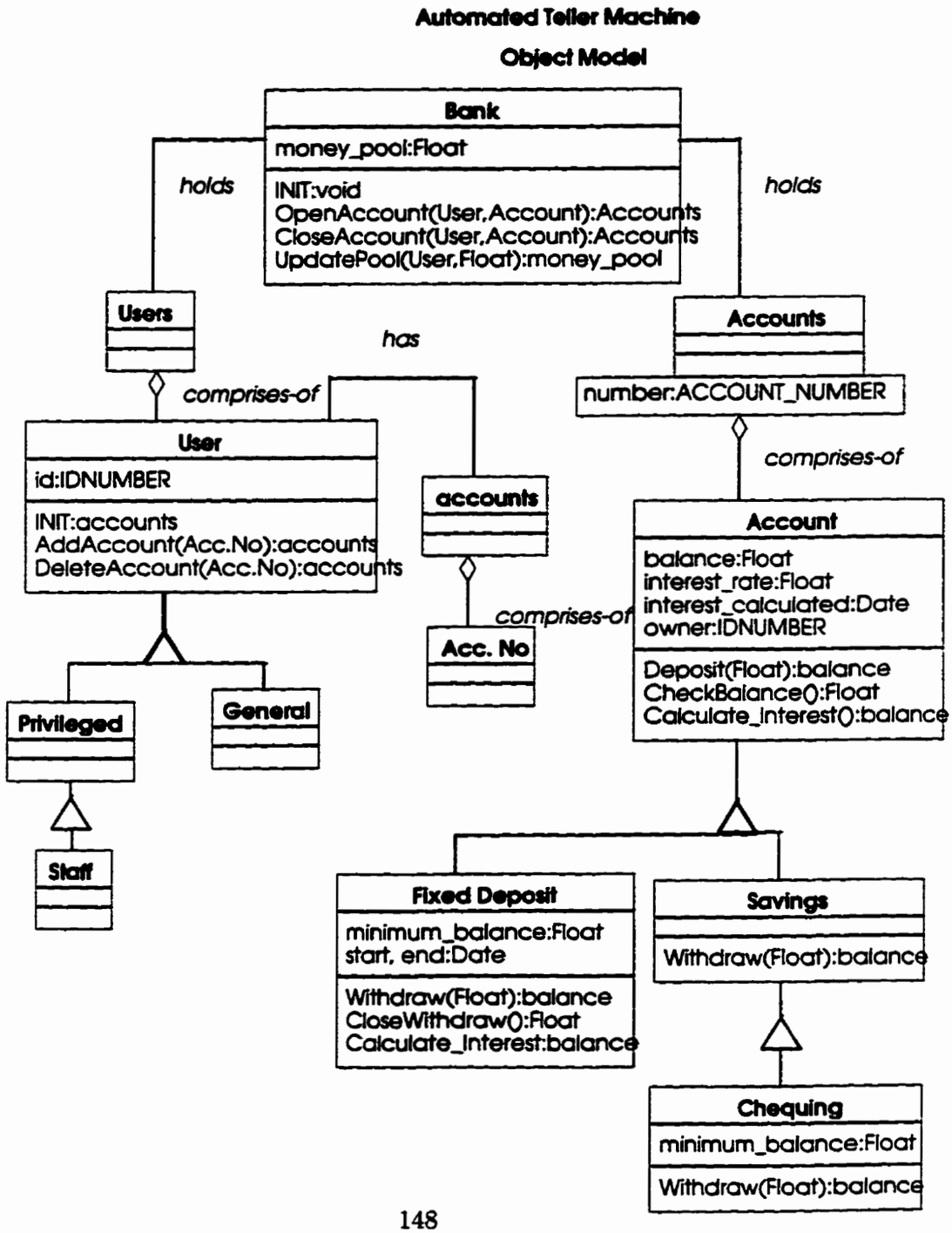


Figure B.1: Object Model for Automated Teller Machine