A COMPARISON OF Z AND UML:
TWO CASE STUDIES

BY

BINGDONG LI

Bachelor of Science
Beijing Agricultural University
Beijing, China
1989

Master of Science
The Graduate School of Chinese Academy of
Agricultural Science
Beijing, China
1996

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 2000

A COMPARISON OF Z AND UML:
TWO CASE STUDIES

Thesis Approved:

Mansur Samadzadeh-H.
Thesis Advisor

Blayne E. Mayfield

J Chandler

Alfred Carlozzi
Dean of the Graduate College

PREFACE

Specification languages usually can provide simple abstract descriptions of complex behaviors of software systems. A large number of specification languages have been introduced for software development. One categorization of specification languages is formal vs. informal. Z and UML are the well-known formal and informal object-oriented specification languages, respectively. In this thesis, related work on Z and UML is reviewed. An overview of Z and UML is also provided. Z and UML are compared from the language aspects based on a set of criteria. In order to provide a practical basis, Z and UML are also compared, based on another set of criteria, using two classic case studies: the library database system and the elevator concurrent control system.

The results of the comparison carried out in this thesis are listed. Z is a formal specification language. Specifications in Z are abstract and are based on mathematical definition. Z is mature and there are lots of tools to support it. UML is an informal object-oriented specification language. UML notations are icons, graphics, and English. Specifications in UML are easy to understand, but have the potential to be ambiguous. UML is getting more popular now and there are lots of tools to support UML. In addition to the theoretical comparison, Z and UML are compared using two classic case studies: the library database system and the elevator concurrent control system. The practical experience revealed that UML provided a good approach to modeling the main concepts using use case diagrams, collaboration diagrams, and state diagrams, while Z provided a

precise model for specification of the problem. Z and UML appear to be complementary. Combining Z and UML in different phases and aspects is the recommended approach.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my advisor Dr. Mansur H. Samadzadeh. He provided guidance through my thesis work. He spent a lot of time reviewing my thesis and providing suggestions for refinement.

I would like to thank my other committee members, Drs. John P. Chandler and Blayne E. Mayfield. Their time and effort are greatly appreciated.

Great thanks also go to my parents and other family members for their understanding and encouragement.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

CHAPTER I

INTRODUCTION

Specification languages provide simple abstract descriptions of complex behaviors of software systems [Cooke et al. 96]. A large number of specification languages have been introduced in software development. Some of these languages target only one phase in the software life cycle, providing the necessary constructs to support a specific abstraction level, while others target more than one phase. One categorization of specification languages is formal vs. informal.

Formal specification languages are based on well-developed mathematical concepts. They support the creation of precise models of system behaviors and reasoning mechanisms. They allow users to derive logical consequences from the explicitly modeled properties. The advantages of formal specification languages can be listed as follows: 1) precision and conciseness of specification can be achieved, 2) mathematical proof procedures can be used to test (and prove) the internal consistency and syntactic correctness of the specifications, and 3) the completeness of a specification can be checked. The drawbacks of formal specification languages can be listed as follows: 1) they require users to be trained in mathematics and in the formal notations, 2) large specifications quickly become unmanageable and unreadable, and 3) they lack the visual and intuitive appeal of graphical techniques. *Z* is one of the well-known formal specification languages.

Informal specification languages usually use a graphical interface along with structures that use some forms of a "natural language". The graphics and the "English-like" nature of these languages make them easy to understand for clients, specifiers, designers, and implementers. However, these informal specification languages tend to be imprecise and ambiguous [France et al. 98]. UML (Unified Modeling Language) is one of the well-known informal specification languages. UML supports object-oriented (OO*)* design and analysis, and was standardized by the Object Management Group (OMG) in 1997 [Booch et al. 99].

The formal method and the object-oriented software development method (OOM*)* are regarded as the two main streams in future software engineering [Lano 90]. In the thesis work, Z was the formal specification language that was used because it is a mature formal specification language, also UML was the informal specification language that was used because it supports OO design and analysis.

Choosing the right modeling language for developing a system may be a difficult task which might influence the cost and qualify of the subsequent development process. When choosing a specification language, some significant and relevant questions can be given as follows. Which is a specification language suitable for a specific project? Can we combine the advantages of both Z and UML? Which software development phase can UML be used for without ambiguities, and which software development phase can Z be used for while retaining understandability? What kinds of projects can UML be used for despite its general lack of precision? The answers to these questions can be used to help decide which specification language is suitable for a particular project.

In the rest of this chapter, the related work of Z and UML is reviewed in Section 1.1. The research objectives are given in Section 1.2, and the organization of this thesis is given in Section 1.3.

1.1 Related Work

In this section, four research trends are briefly reviewed as follows. The comparison of specification languages is reviewed in Section 1.1.1. The formalization of UML is reviewed in Section 1.1.2. The visualization of Z is reviewed in Section 1.1.3, and the combination of Z and UML is reviewed in Section 1.1.4.

1.1.1 Comparison of Specification Languages

Many specification languages have been developed and are in use today. For a particular application, it is hard to choose the best language because there are significant differences among these languages. Several research efforts have attempted to compare two or more specification languages based on various criteria.

One way to compare specification languages is to use them to specify well-known problems. Numerous specifications exist for the library problem, and twelve of them were compared by Wing [Wing 88]. Four of these 12 specifications used informal approaches, and the other eight used formal approaches. Wing discussed the 12 specifications with respect to these criteria: software life cycle phase(s) applied, modularity, readability, ambiguity, and incompleteness. This paper showed that benefits could be obtained by properly applying formal or informal specification methods.

Ardis et al. [Ardis et al. 96] used Modechart, VFSM, ESTEREL, Basic LOTOS, SDL, Z, and C to specify AT&T's 5ESS® telephone switching systems. These specification languages were compared by the authors based on several criteria that were classified as fundamental and important. Their fundamental criteria included applicability, usability in embedded system, implementability, maintainability, modularity, level of abstraction/expressibility, verifiability, and tool maturity. Their important criteria included looseness, learning curve, language maturity, data modeling, and discipline.

A formal specification language, Knowledge-Based System (KBS), was evaluated under the following criteria: expressibility, errors in the formalization process, redundancy and repetition, locality of changes, reusability, and guidelines [Harmelen et al. 96]. These criteria can also be used to compare other specification languages.

Bicarregui and Ritchie [Bicarregui and Ritchie 95] compared the VDM and B notations through a communication protocol case study with respect to these criteria: invariance, operation definitions, and framming. Another study compared OPEN and UML with respect to process and lifecycle support, meta-model, and notation [Henderson-Sellers and Firesmith 99].

1.1.2 Formalization of UML

While UML reflects some of the best OO modeling experiences available [Booch et al. 99], it suffers from a general lack of precise semantics that is necessary if one is to use the UML notation to model systems precisely and to reason rigorously about models. Several efforts have tried to formalize UML [Bourdeau and Cheng 95] [France et al. 98]. Two general approaches to formalizing OO modeling concepts were supplemental [Cook and Daniel 94] and OO-extended formal languages [Lano 90].

In the supplemental formal language approach, an elaborate example was given by Cook and Daniel [Cook and Daniel 94]. In their paper, the semantics of the static data model was also defined.

In the OO-extended formal language approach, Lano extended an existing formal notation with OO features [Lano 90]. From a practical perspective, a problem with the notations developed using this approach is lack of analysis tool. Also, the models created using these notations are not easy to read, understand, and modify because of the large semantic gap between real world concepts and their mathematical representations in the formal notations [France et al. 98].

Shroff and France [Shroff and France 97] demonstrated a method to formalize primary UML constructs using Z notation. Also, a precise semantics for UML was proposed by France [France et al. 98].

1.1.3 Visualization of Z

Z has shown its ability to improve the quality of software systems by reducing possible faults in various ways [Bowen 96]. The most often stated limitations of Z are difficulties in using and understanding its formal notation. Various graphical notations have been used to visualize formal specifications written in Z [Kim and Carringto 99]. It seems that the scope of their visualization was limited, since it is hard to visualize all aspects of a Z specification.

1.1.4 Combination of Formal and Informal Specification Languages

Work on the combination of OO and formal methods appeared in the 1990's. Informal modeling techniques are made more precise and amenable to rigorous analysis by combining them with a suitable formal specification notation. A number of efforts have been proposed [France et al. 97]. Bourdeau and Cheng [Bourdeau and Cheng 95] presented formal semantics for Rumbaugh's OMT Object diagrams using Larch as the formal notation. Researchers at Swinburne University, Australia, have proposed an integration of MOSES, a graphical OO modeling notation, with Object-Z called FOOM (Formal Object-Oriented Modeling) [Henderson-Sellers and Edwards 94]. These works focused on the generation of formal specification from less formal OO models, and focused on mapping UML primary constructs to Z primary constructs.

1.2 Research Objectives

The objectives of this thesis work included comparing Z and UML in different domains, and finding a suitable mode for using Z and UML. The contributions of this work are that the research results can help software specifiers to decide which of Z and UML is more appropriate for a given problem depending on the characteristics of the problem, and for which software development phase it is more appropriate to use Z or UML. It also can provide background information for designers who will design new specification languages or design tools for either formalizing UML or visualizing Z.

1.3 Organization of the Thesis

The rest of this thesis report is organized as follows. An overview of Z and UML is provided in Chapter II. A comparison of Z and UML is provided in Chapter III. Two case studies are given in Chapter IV. Summary and suggested future work are provided in Chapter V.

CHAPTER II

OVERVIEW OF Z AND UML

Z and UML are overviewed in Sections 2.1 and 2.2, respectively. A brief history, basis, basic elements, brief evaluation, and development processes of Z and UML are discussed.

2.1 Overview of Z

2.1.1 Brief History

The specification language Z was originally developed by Jean-Raymond Abrial. It evolved from research at the Oxford University Computing Laboratory (OUCL) in 1970's [Schach 97].

2.1.2 Basis

Z is a model-based specification language that relies strongly on set theory and first order predicate logic [Hayes 87]. Mathematical theories such as set theory, relations, functions, sequences, bags, and models are its foundations.

2.1.3 Basic Elements

A specification in Z consists of a sequence of definitions for global set names, global variables, and schemas. A global set name is an entity of the system defined by users or by Z. A global variable is an object of entities and is used in schemas. A schema is a collection of variable declarations (or signatures) and predicates for relationships among the variables (Figure 2.1). The rules of operations on schemas are inheritance, composition, and information hiding. Pre-conditions and post-conditions typically are not labeled explicitly. Z describes both the state space and the transitions on the states, and places no restrictions on the style of specifications.

The basic set operations in Z are set membership (∈), set union (∪), domain subtraction (⩤), and function override (⊕).

Variables may end with "?", indicating that the variable is to be input, or with "!", indicating that the variable is to be output.

By convention, the unprimed variables stand for the state before the operations and the primed variables stand for the state after the operation. Placing the delta symbol (Δ) in front of an existing schema name, indicates that the operation of the schema will change some of the variables of the state.

The Z notations can be found in Appendix B.

```
┌---------Schema Name------------------------------------------------ (Schema box)
| variable declarations
|------------------------------------------------- (Such that)
| predicates
|
---------------------------------------------------------------------------------------------
```

Figure 2.1 Schema Definition in Z

2.1.4 Evaluation

Specifications written in Z are precise, unambiguous, concise, and amenable to proof. Z seems to be well suited for defining modules at the requirement analysis level [Neil et al. 98]. It emphasizes both presentation and specification reuse. A specification written in Z typically needs informal explanatory comments in a natural language such as English to make it more readable and understandable.

Z was originally designed to be read by humans rather than by computers [Bowen 96], thus it is not executable. Z is not ideal for all problems.

Z has been used in the academia and the industry. For example, IBM Customer Information Control System (CICS) was specified using Z [Bowen 96]. Many academic examples are also available [Hayes 87].

2.1.5 Development Process

Specifications written in Z usually involve specifying the problem in both English and Z. They are usually composed of global entities, states of the system, and operations of the system. The two case studies in Chapter IV follow this process.

In designing a state model, software engineers look for the minimum number of state variables that describe states of the system and operations on them. The types of variables also need to be identified. A type is a collection of objects. Types are similar to sets, in the sense that they are collections of values that share similar characteristics.

There are a number of standard types in Z such as the type of natural numbers. It is also possible to introduce user-defined types for specific systems.

A state is a collection of sets and objects, and some predicates defined on those entities. The choice of state model depends on the functions the system is expected to perform.

Operations are the functions of the system that need to be carried out step by step. Each operation on the systems states has to be specified. Operations are usually specified using schemas with names, declarations, and predicates.

2.2 Overview of UML

2.2.1 Brief History

The Unified Modeling Language (UML) was initially created by the merger of ideas mainly from three approaches [Booch et al. 99]: Booch, Jacobson's Object-Oriented Software Engineering (OOSE*)*, and Rumbaugh's Object Modeling Technique (OMT). The critical mass of ideas started to form by the mid 1990s, and it was standardized in 1997.

2.2.2 Basis

UML is an informal specification language. It consists of meta-models and notations [Booch et al. 99]. It is built on OO concepts.

2.2.3 Basic Elements

There are three basic elements in UML [Booch et al. 99]. They are: UML's basic building blocks, the rules that put basic building blocks together, and some common mechanisms that apply throughout UML.

2.2.3.1 Building Blocks

There are three kinds of building blocks in UML: thing, relationship, and diagram. Things in the UML include structural things, behavioral things, grouping things, and annotational things. Structural things are the nouns of UML models. They are class, interface, collaboration, use case, active class, component, and node (see Appendix C). A class is a description of a set of objects that share the same attributes and operations. An interface is a collection of operations that specify a service of a class or a component. A collaboration defines an interaction and is a society of roles and other elements that work together to provide some cooperative behaviors. A use case represents how to use the system. An active class is just a class except that its objects represent elements whose behavior is concurrent with other elements. A component is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. A node is a physical element that exists at run time and represents a computational resource. Behavior things are the verbs of UML models. They include interaction and state machine (see Appendix C). An interaction is a behavior in which a set of objects exchange messages. A state machine is a behavior that specifies the sequences of states of an object or an interaction. A package is the grouping thing that groups elements together. A note is the annotational thing that explains UML models in English.

Relationships include dependency, association, generalization, and realization. A dependency is a relationship between two things in which a change to one thing may affect the other thing. An association is a relationship that describes a set of links among objects. A generalization is a specification/generalization relationship that generalizes a super-class from a subclass or specializes a subclass from a super-class. Realization is a semantic relationship between classifiers, that is, one classifier specifies a contract that another classifier guarantees to realize.

Diagrams group interesting collections of things, and are the graphical presentation of a set of elements. There are nine kinds of diagrams: class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, statechart diagram, activity diagram, component diagram, and deployment diagram.

2.2.3.2 Rules

UML has semantic rules for name, scope, visibility, integrity, and execution in order to be understood by different people.

2.2.3.3 Mechanisms

Common mechanisms in UML are: specification, adornment, common division, and extensibility. Specifications provide a textual statement of the syntax and semantics of the building blocks. Adornments specify the notation with additional meanings. Common divisions distinguish the things in different ways.

Extensibility mechanisms include stereotype, tagged value, and constraint. A stereotype extends the vocabulary of UML, allowing users to create new kinds of building blocks that are derived from the existing ones but that are specific to the user's problem. A tagged value extends the properties of a UML building block, allowing users to create new information. A constraint extends the semantics of a UML building block, allowing users to add new rules or modify an existing one.

2.2.4 Evaluation

The original developers of UML usually introduce it with claims of wide-ranging applicability as follows [Booch et al. 99].

> The UML is appropriate for modeling systems ranging from enterprise information systems to distributed Web-based applications and even to hard real time embedded systems.

However, UML is an informal specification language. It still suffers from a general lack of precision [Shroff and France 97]. Developers may waste considerable time resolving disputes over the usage and interpretation of the notation. Rigorous semantic analysis is difficult. These problems have been widely recognized [France et al. 98].

UML is not executable. However, there are several CASE tools that support UML, the most well-known one is Rational Rose [France et al. 98].

2.2.5 Development Process

The UML standard specifies a notation for several different diagrams, but it does not describe how to create and apply each diagram. There are three well-known software processes in UML [Booch et al. 99]. The first one is use case driven, the second one is architecture-centric, and the third one is iterative and incremental. In the use case driven process, use cases are the primary artifacts of establishing the desired behavior of the system, for verifying and validating the system's architecture, for testing, and for communicating among the stakeholders of the project. In the architecture-centric process, the system's architecture is used as the primary artifact for conceptualizing, constructing, and managing the system under development. In the iterative and incremental processes, each new release is focused on attacking and reducing the most significant risks to the success of the project.

These three processes (the use case driven process, the architecture-centric process, and the iterative and incremental process) can be structured along two dimensions: time and process components. Structuring a project along the time dimension includes the following time-based phases: inception, elaboration, construction, and transition. In the inception phase, the project version is specified. In the elaboration phase, the necessary activities and the required resources are planed, the features of the system are specified, and the architecture is designed. In the construction phase, the product is built as a series of incremental iterations. In the transition phase, the product is supplied to the user community. Structuring the project along the process component dimension includes the following activities: requirement capture, analysis and design, implementation, and test. Requirement capture is a narration of what the system should do. Analysis and design is a description of how the system will be realized. Implementation is the product of the code that will result in an executable system. Test is the verification of the entire system.

This thesis report concentrates on the use of UML as a specification language to capture requirement and analysis, and to design during the inception and elaboration phases of software development. The three subsections that follow outline a brief introduction of the process that will be used by the two case studies included in Chapter IV.

2.2.5.1 Creating Use Cases

The use case model starts in the inception phase with the identification of actors and principal use cases for the system. Use cases model diagrams between actors and the system. Actors represent anyone or anything that must interact with the system. They are usually not part of the system. The collection of use cases for a system constitutes all the defined ways the system may be used. A use case diagram is a graphical view of some or all of the actors, the use cases, and their interactions identified for a system. Each system typically has a main use case diagram. Other use case diagrams may be created as needed. Documentation of a use case includes main course, alternative course, precondition and post-condition. Sequence diagram scenarios describe the interactions between the external actors and the system.

2.2.5.2 Identifying Behaviors

There are two types of interaction diagrams: sequence diagram and collaboration diagram. A sequence diagram shows object interactions arranged in time sequence. Sequence diagrams typically are associated with use cases in the model of the software system under development. Boundary classes are added to sequence diagrams to show the interaction with a user or with another system. A collaboration diagram shows object interactions organized around the objects and their links to each other. In syntax, sequence diagrams and collaboration diagrams are analogous and can be transferred to each other. In this thesis report, collaboration diagrams will be emphasized over sequence diagrams because of their ability to convey more contextual information, and their relative spatial economy. State diagrams are another diagrams of UML to describe the dynamic aspects of a system. State diagrams describe a system changing from one state to another state or from one activity to another activity.

2.2.5.3 Checking the Model

In order to make the model homogeneous, great care needs to be paid. Classes may need to be combined, split, or eliminated. New methods may need to be added later on. Consistency of all diagrams needs to be checked.

CHAPTER III

A COMPARISON OF Z AND UML

In this chapter, the formal specification language Z and the object-oriented specification language UML are compared from various language aspects. The comparison criteria are introduced in Section 3.1. The comparison results and detailed explanation are given in Section 3.2.

3.1 Comparison Criteria

Numerous comparison criteria exist to compare different specification languages from various aspects [Wing 88] [Ardis et al. 96] [Harmelen et al. 96] [Bicarregui and Ritchie 95] [Henderson-Sellers and Firesmith 99]. The criteria used to compare Z and UML in this chapter are: formality, life-cycle phases applied, modularity, notation, specification style, tool support, language maturity, and orientation. The description of each criterion follows.

Formality: A specification language is formal if it has a precise and unambiguous syntax. A precise and unambiguous syntax is given by mathematical definitions usually in the form of a set of definitions, a set of logical formulas, or an abstract model.

Life-cycle Phases Applied: The life-cycle phases applied by a specification language can be one or more of these phases: specification, analysis, design, and detailed design.

Modularity: Whether or not the method uses simple composition operators that allow a large specification to be more easily written and understood by decomposing it into smaller parts. Whether or not it is possible to make conceptually small additions, changes, and generalizations to a specification without major rewriting.

Notation: Notation impacts the communication among software developers, domain experts, users, customers, managers, and quality assurance personnel.

Specification Style: Specification style refers to the main elements of the specification and the appearance of the specification.

Tool Support: Tool support is an indication of the popularity of a language in industry. The more the number of tools, the more popular the specification language is. Users can have more choices when they search for support tools.

Language Maturity: How much development has the language undergone in terms of certification, amount and quality of training available, user base size, and industrial deployment.

Orientation: Orientation refers to what the process is based on and what rules the process follows (e.g., data orientation, operation orientation, or control orientation).

3.2 Comparison of Z and UML

In the following subsection, Z and UML are compared based on the criteria identified in Section 3.1.

3.2.1 Formality

Z is a formal specification language because it uses set theory, first order predicate logic, functions, etc.

UML is an informal specification language because UML uses graphical notations and plain English. UML uses stereotypes to extend vocabularies in order to specify mathematical operations, but it still has the potential to be ambiguous.

3.2.2 Life-Cycle Phases Applied

Experiences in Z specifications have showed that Z is suitable for requirement specifications [Hayes et al. 93]. It is very helpful when proof is needed in the design phase.

UML is largely process independent, and it is claimed that it is suitable for any software development life cycle phase [Booch et al. 99]. However, it has been argued that UML is not suitable for all life cycle phases for business issues [Henderson-Sellers and Firesmith 99].

3.2.3 Modularity

The structuring of functionality between Z schemas is arbitrary [Neil et al. 98]. In particular, schemas are frequently tightly coupled, have weak cohesion, and have duplicate functionality. Z schemas are more like macros than models, so the body of a schema may refer to items that are not declared directly in the schema. So, specifications in Z generally seem clumsy when Z is used to specify large systems.

UML is based on meta-models, graphical notations, and OO concepts. These characteristics make UML modular and easier to use and understand for a large specification.

3.2.4 Notation

The Z notation is provided in Appendix B. Z uses mathematical notation and remains rather strict. The advantages are that the mathematical notations have precise and defined semantics. Z notations are easy to understand for people who have a strong mathematical background. For other people, they are hard to understand.

The UML notation is provided in Appendix C. UML uses graphical notation with plain English explanation. The UML notation is easy to understand. A disadvantage is that the notation has the potential to be ambiguous.

3.2.5 Specification Style

Specifications in Z typically consist of definitions and schemas. Schemas are used to describe not only states but also operations. Schemas are used as building blocks to construct descriptions of states. A specific target programming language is needed when specifications are refined for implementation.

Specifications in UML consist of graphical icons, diagrams, lines, and plain English. Classes and objects are the basic building blocks. They are used to draw different diagrams with relationships, time, and dynamic states. They can also be grouped together as packages. There is no specific target programming language needed when specifications in UML are refined for implementation.

3.2.6 Tool Support

Tool support for Z and UML are listed in Appendix D and Appendix E, respectively. Both Z and UML have lots of support tools available. Most of Z support tools follow the standard Z notation and force users to obey the standard rules. In contrast, most of UML support tools follow different notations. These tools are generally not compatible.

3.2.7 Language Maturity

Z was developed in the 1970's. A reference manual for Z was published in 1989 [Spivey 89], and it was recognized as the standard Z notation.

Object-oriented modeling languages appeared in mid 1970's and 1980's. The critical ideas of UML started in mid 1990's. The UML effort started officially in October

1994, and version 0.8 draft of UML was released in October 1995. There are still lots of researches trying to make UML more formal by using Z.

3.2.8 Orientation

Z is operation oriented. Schemas are the basic building blocks that are used to specify operations. Typically, given a system to be modeled using Z, the system variables are identified and treated as globally accessible, also the system functions must be identified and the behaviors of the functions are either left unspecified or described through simplistic models. Initial conditions are typically stated explicitly.

UML is data oriented. Class is the basic OO data type. A system's functionality is either left unspecified or indirectly described through interactions of operations of the different data types. Initial conditions are typically stated implicitly through creating operations on objects.

CHAPTER IV

TWO CASE STUDIES

In this chapter, two case studies are discussed in order to lend some practical perspective to the comparison of Z and UML. A brief introduction of the comparison criteria for the two case studies is provided in Section 4.1. The library problem is given in Section 4.2. The elevator control problem is given in Section 4.3.

4.1 Introduction of Comparison Criteria

The comparison criteria used to compare specifications in Z and UML in this chapter are: readability, applicability, levels of abstraction / expressibility, verifiability, discipline, redundancy, reusability, and ambiguity. The descriptions of these criteria are given below.

Readability: Whether or not graphics or diagrams are used to aid readability.

Applicability: Whether or not the technology describes real world problems and solutions in a natural and straightforward manner.

Levels of Abstraction / Expressibility: How closely and expressively the specification can describe, from the user's point of view, the objects and the environment in the domain. How large the ratio is between the amount of interesting properties of a system and the amount of extra information.

Verifiability: Whether or not it is possible to demonstrate formally that a specification satisfies the properties stated at the same or a higher level of abstraction.

Discipline: Whether or not the language forces users to write reasonably well-behaved specifications.

Redundancy: Whether or not a piece of knowledge is represented more than one way in the same model.

Reusability: Whether or not we can reuse parts of a model specification in formalizing other models.

Ambiguity: Whether or not a specification specifies the relevant issues in a given problem precisely.

4.2 Library Problem Case Study

An introduction to the problem is given in Section 4.2.1. The Z approach and the UML approach are given in Sections 4.2.2 and 4.2.3, respectively. A comparison of these specifications is provided in Section 4.2.4.

4.2.1 Introduction to the Library Problem

The library problem is a classic problem in software specification [Wing 88]. It was first discussed by Kemmerer in 1981 as part of his class lecture on formal specification at the University of California at Santa Barbara. Later on several authors used it as a case study in their teaching or research. The problem attained broader prominence in 1986 when it appeared in the Fourth International Workshop on Software Specification and Design as one of a set of four problems.

The requirement of the library problem is stated as follow [Wing 88].

> Consider a small library database with the following transactions.
>
> (1) Check out a copy of a book /return a copy of a book.
> (2) Add a copy of a book to / remove a copy of a book from the library.
> (3) Get the list of books by a particular author or in a particular subject area.
> (4) Find out the list of books currently checked out by a particular borrower.
> (5) Find out what borrower last checked out a particular copy to a book.
>
> There are two types of users: staff users and ordinary borrowers. Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints.
>
> (1) All copies of the books in the library must be available for check out or be checked out.
> (2) No copy of a book may be both available and checked out at the same time.
> (3) A borrower may not have more than a predefined number of books checked out at one time.

4.2.2 Z Approach

4.2.2.1 Specification in Z

This specification does not intend to show all aspects of Z. It is the author's view of the problem and it has been modified from the specifications given by Diller [Diller 90] and King and Sorensen [King and Sorensen 89]. Z notations and textual English are mixed in order to explain the problem more clearly. The development process follows a process of which an overview was presented in Section 2.1.5. Global entities, states of the system, and operations of the system are the three main components.

4.2.2.1.1 Global Entities

This specification makes use of the following given types: *[Book, Copy, Person, Author, Subject, Report, MaxCopiesAllowed]* as explained below. *Book* is the set of all possible books. Books are considered to be abstract objects. *Copy* is the set of all possible copies of books, and is a physical object. *Person* is the set of all possible people in this system. *Author* is the set of all possible authors including institutions. *Subject* is the set of all conceivable subjects that books can be about. *Report* is the set of all necessary messages. *MaxCopiesAllowed* is the maximum number of copies of books anyone can borrow.

```
[PERSON]
    staff: P PERSON
    borrower: P PERSON
    staff I  borrower = { }
[COPY]
    stock: P COPY
    available: P COPY
    checked-out: COPY ⇸ PERSON
    (dom checked-out) Y available = stock
    (dom checked-out) I  available = { }
    ∀ p: PERSON • # (checked-out ) > { p} ≤ MaxCopiesAllowed
    last-checked-out: COPY ⇸ PERSON
    checked-out ⊇ last-checked-out
    (ran checked-out) ⊆ (staff Y borrower)
```

For the type *Book*, the subset *[Title, Author, Subject]* is introduced.

```
------Book---------------------------------------------------------------------------
title: Title
authors: F Author
subject: F Subject
--------------------------------------------------------------------------------------
```

Report = 'Okay'
| 'Unauthorized requestor'
| 'Not registered'
| 'Copy checked out'
| 'Cannot borrow any more copies'
| 'Copy available'
| 'This book is not new to the library'
| 'This book is new to the library'
| 'This copy is owned by the library'
| 'This book is not owned by the library'
| 'This is the only copy in the library'
| 'This is not the only copy in the library'
| 'Unknown borrower'
| 'Copy not previously borrowed'

4.2.2.1.2 States of the System

The library system is split as *LibrarySystem*, *LibraryDB*, and *Library* states. Each state is specified as follows.

```
---------LibrarySystem-------------------------------------------------------------
| instanceof: Copy ⇸ Book
| writtenby : Book ⇸  F Author
| about: Book ⇸    F Subject
|---------------------------------------------------------------------
| dom writtenby ⊆ ran instanceof
| dom about ⊆ ran instanceof
-------------------------------------------------------------------------------------------
```

The *LibrarySystem* schema contains information relating to the books in the library. The function *instanceof* tells which book a copy is an instance of. The function *writtenby* tells who writes a particular book. It returns a set of authors that may be empty. Some books may be journals associated with the empty set of authors. The function *about* tells what a book is about. The predicate *dom writtenby ⊆ ran instanceof* tells us that the database does not contain author information about books that are not in the library, and the predicate *dom about* ⊆ *ran instanceof* tells us that the database does not contain subject information about books which are not in the library.

-----------*LibraryDB*---
borrower, staff: F person
available, checked-out: F Copy
previouslyborrowedby, borrowedby: COPY ⇸ *PERSON*
----
borrower I *staff* = *{ }*
available I *checked-out* = *{ }*
dom borrowedby = *checked-out*
ran borrowedby ⊆ *borrower*
dom previouslyborrowedby ⊆ *available* Y *checked-out*
ran previouslyborrowedby ⊆ *borrower*
$\forall\ p\colon Person \mid p \in borrower \bullet \#\, borrowedby^{-1} (\mid \{p\} \mid) \leq MaxCopiesAllowed$

The finite sets of *borrower* and *staff* contain people who can use the library and those who manage it, respectively. The predicate *borrower* I *staff* = *{ }* tells that the identifiers of borrowers and staff are disjoint.

The finite sets of *available* and *checked-out* contain all the copies of the books that are either available for checkout or have actually been checked out. The predicate *available* I *checked-out* = *{ }* captures the following constraint from the statement of requirements:

- No copy of a book may be both available and checked out at the same time.

The function *borrowedby* tells us who has currently borrowed a particular copy of a book, and the function *prevouslyborrowedby* tells us who is the last person borrowing a copy of a book. For example, let c be a particular copy of a book that is currently on loan. Then, *borrowedby(c)* is the current borrower of c, and *previouslyborrowedby(c)* is the person, if any, who borrowed c immediately before the person *borrowedby(c).*

The predicate *dom borrowedby* = *checked-out* tells us that the set of copies that have been borrowed by people is the same as the set of copies that are checked-out. The predicate *ran borrowedby* $\subseteq$ *borrower* states that every *checked-out* copy must have been borrowed by a *borrower*.

The predicate *dom previouslyborrowedby* $\subseteq$ *available* Y *checked-out* states that previously borrowed copies are either available for checkout or have been checked out again. The predicate *ran previouslyborrowedby* $\subseteq$ *borrower* states that the last borrower of every copy must be a registered user. The final predicate $\forall$ *p: Person* | *p* $\in$ *borrower* • *# borrowedby* $^{-1}$ *(| {p} |)* $\leq$ *MaxCopiesAllowed* captures the constraint:

- A borrower may not have more than a predefined number of books checked out at one time.

The complete state of the library is given by means of *Library*:

```
----------------------Library-----------------------------------------------------------
| LibrarySystem
| LibraryDB
|--------------------------------------------------
| dom instanceof = available Y checked-out
-----------------------------------------------------------------------------------------
```

dom instanceof = *available* Y *checked-out* tells us that the database does not contain information about copies of books which are not in stock.

The schemas Δ *LibrarySystem*, Δ *LibraryDB*, and Δ *Library* are defined in the usual way:

Δ *LibrarySystem* $\cong$ *LibrarySystem* $\wedge$ *LibrarySystem'*
Δ *LibraryDB* $\cong$ *LibraryDB* $\wedge$ *LibraryDB'*
Δ *Library* $\cong$ *Library* $\wedge$ *Library'*

The schemas $\Xi\, LibrarySystem$, $\Xi\, LibraryDB$, and $\Xi\, Library$ are defined in the standard way:

$$\begin{array}{ll} \Xi\, LibrarySystem \cong & \Delta\, LibrarySystem \\ & \mid instanceof' = instanceof \\ & \wedge\, writtenby' = writtenby \\ & \wedge\, about' = about \end{array}$$

$$\begin{array}{ll} \Xi\, LibraryDB \cong & \Delta\, LibraryDB \\ & \mid borrower' = borrower \\ & \wedge\, staff' = staff \\ & \wedge\, available' = available \\ & \wedge\, checked\text{-}out' = checked\text{-}out \\ & \wedge\, previouslyborrowedby' = previouslyborrowedby \\ & \wedge\, borrowedby' = borrowedby \end{array}$$

$$\Xi\, Library \cong \Xi\, LibrarySystem \wedge \Xi\, LibraryDB$$

In the initial state, every variable is an empty set.

$$\begin{array}{ll} InitialLibrarySystem' \cong & LibrarySystem' \\ & \mid instanceof' = \{\,\} \\ & \wedge\, writtenby' = \{\,\} \\ & \wedge\, about' = \{\,\} \end{array}$$

$$\begin{array}{ll} InitialLibraryDB' \cong & LibraryDB' \\ & \mid borrower' = \{\,\} \\ & \wedge\, staff' = \{\,\} \\ & \wedge\, available' = \{\,\} \\ & \wedge\, checked\text{-}out' = \{\,\} \\ & \wedge\, prevouslyborrowedby' = \{\,\} \\ & \wedge\, borrowedby' = \{\,\} \end{array}$$

$$InitialLibrary' \cong InitialLibrarySystem' \wedge InitialLibraryDB'$$

4.2.2.1.3 Operations of the System

There are eight operation models in this system. They are described one by one as follows.

1) Checking-out Copies of Books

---------*CheckOutCopy*---
$\Delta\ Library$
$\Xi\ LibrarySystem$
$n?: Person$
$c?: Copy$

$n? \in borrower$
$c? \in available$
$\#borrowedby^{-1}\ (\mid \{\ n?\ \} \mid) < MaxCopiesAllowed$
$available' = available \setminus \{\ c?\ \}$
$checked\text{-}out' = checked\text{-}out\ \mathrm{Y}\ \{\ c?\ \}$
$borrowedby' = borrowedby\ \mathrm{Y}\ \{\ c? \rightarrow n?\ \}$

$previouslyborrowedby' = previouslyborrowedby$
$borrower' = borrower$
$staff' = staff$

The preconditions of the schema *CheckOutCopy* are the three predicates: $n? \in borrower$, $c? \in available$, and $\#borrowedby^{-1}\ (\mid \{\ n?\ \} \mid) < MaxCopiesAllowed$. The predicate $n? \in borrower$ states that only registered users can borrow copies. The predicate $c? \in available$ states that only available copies can be borrowed. . The predicate $\#borrowedby^{-1}\ (\mid \{\ n?\ \} \mid) < MaxCopiesAllowed$ states that a borrower can only borrow copies if the number of copies he currently has borrowed is strictly less than the maximum number allowed.

The following schema, *AuthorisedRequestor,* is used to record the fact that the person *requestor?* is allowed to perform those transactions which only staff members of the library can perform.

--------*AuthorisedRequestor*---
$staff: F\ Person$
$requestor?: Person$

$requestor? \in staff$

The successful operation of checking out a copy is specified by the schema:

AuthorisedRequestor ∧ *CheckOutCopy*

The schema *UnauthorisedRequestor* specifies what happens when an unauthorized person tries to perform the transaction of checking out a copy of a book. The state of the database is unchanged, but an error message is displayed.

```
|--------------UnauthorisedRequestor---------------------------------------------
| Ξ Library
| requestor?: Person
| rep!: Report
|-------------------------------------------
| requestor? ∉ staff
| rep! = 'Unauthorized requestor'
|--------------------------------------------------------------------------------
```

The schema *Unregistered* specifies what happens when someone not registered with the library tries to borrow a copy.

```
|--------------Unregistered------------------------------------------------------
| Ξ Library
| n?: Person
| rep!: Report
|---------------------------------
| n? ∉ borrower
| rep! = 'Not registered'
|--------------------------------------------------------------------------------
```

The schema *CopyNotOwned* specifies what happens when someone tries to borrow a copy that is not in the library, meaning that the library does not own a copy.

```
|--------------CopyNotOwned------------------------------------------------------
| Ξ Library
| c?: Copy
| rep!: Report
|-------------------------------------------------------
| c? ∉ available Y checked-out
| rep!: = ' This copy is not owned by the library'
|--------------------------------------------------------------------------------
```

The schema *CopyCheckedOut* specifies what happens when someone tries to borrow a copy that is owned by the library but currently has been checked out.

-----*CopyCheckedOut*-----

Ξ *Library*
c?: Copy
rep!: Report

$c? \in checked\text{-}out$
$rep! = \text{'Copy checked out'}$

The schema *TooManyCopies* specifies what happens when someone tries to borrow a copy of a book when he has already borrowed the maximum number of copies he is allowed to.

------*TooManyCopies*-----

Ξ *LibraryState*
n?: Person
rep!: Report

$n? \in borrower$
$\#borrowedby^{-1} (| \{ n? \} |) = MaxCopiesAllowed$
$rep! = \text{'Cannot borrow any more copies'}$

The schema *Success* just outputs a confirmatory message that the operation being performed has been successfully completed.

----------------*Success*-----

rep!: Report

$rep! = \text{'Okay'}$

Thus, the complete specification of the transaction of borrowing a copy of a book is captured by the schema *DoCheckOutCopy:*

$$\begin{aligned} DoCheckOutCopy \cong\ & AuthorisedRequestor \\ & \wedge\ CheckOutCopy \\ & \wedge\ Success \\ & \vee\ UnauthorisedRequestor \\ & \vee\ Unregistered \\ & \vee\ CopyNotOwened \\ & \vee\ CopyCheckedOut \\ & \vee\ TooManyCopies \end{aligned}$$

2) Returning a Copy of a Book

---------*Return*---

$\Delta\ Library$

$\Xi\ LibrarySystem$

$c?: Copy$

$c? \in checked\text{-}out$

$available' = available\ \mathrm{Y}\ \{ c? \}$

$checked\text{-}out' = checked\text{-}out \setminus \{ c? \}$

$borrowedby' = \{ c? \} ⩤ borrowedby$

$previouslyborrowedby' = previouslyborrowedby \oplus \{ c? \rightarrow borrowedby (c?) \}$

$borrower' = borrower$

$staff' = staff$

The precondition of *Return* is the single predicate $c? \in checked\text{-}out$. The post-condition is captured by the middle group of four predicates. The first two of these state that the copy *c?* is transferred from actually being borrowed to just being available for loan. The third removes the information relating to *c?* from *borrowedby'*, and the fourth predicate records the information that the last person to borrow *c?* was *borrowedby (c?)*, overriding any information about who previously borrowed *c?*.

The schema *CopyAvailable* captures that the copy is available for the library.

-------------*CopyAvailable*--

Ξ *LibraryState*
c?: Copy
rep!: Report

$c? \in available$
rep! = 'Copy available'

The total specification of the transaction of returning a copy of a book is given by means of the schema *DoReturn*.

$$
\begin{aligned}
DoReturn \cong\; & AuthorisedRequestor \wedge Return \wedge Success \\
& \vee\; UnauthorisedRequestor \\
& \vee\; Unregistered \\
& \vee\; CopyNotOwned \\
& \vee\; CopyAvailable
\end{aligned}
$$

3) Adding a Copy of a Book to the Library

In adding a copy of a book to the library, we need to consider two cases: when the book is new to the library and when the library already owns a copy of the book.

Adding a New Book to the Library

One of the preconditions of *AddNewBook* is $b? \notin ran\ instanceof$. This precondition states that we can only add an entirely new book to the library if the library does not contain this book. The situation in which it is attempted to add a new book to the library, but the book is not new to the library, is captured by the schema *NotNewBook*.

```
------AddNewBook---------------------------------------------------------------
 ΔLibrary
 c?: Copy
 b?: Book
 A?: F Author
 S?: F Subject
-------------------------------------------------------
 b? ∉ ran instanceof
 c? ∉ available Y checked-out

 available' = available Y { c? }
 instanceof' = instanceof Y { c? → b? }
 writtenby' = writtenby Y { b? → A? }
 about' = about Y { b? → S? }

 checked-out' = checked-out
 previouslyborrowedby' = previouslyborrowedby
 borrowedby' = borrowedby
 borrower' = borrower
 staff' = staff
------------------------------------------------------------------------------------------
```

```
--------NotNewBook---------------------------------------------------------------
 ΞLibrary
 b?: Book
 rep!: Report
-------------------------------------------------------
 b? ∈ ran instanceof
 rep! = 'This book is not new to the library'
------------------------------------------------------------------------------------------
```

The precondition *c? ∉ available* Y *checked-out* states that the copy *c?* that we are adding to the library must not already be owned by the library. This situation is captured by the schema *CopyOwned*.

```
----CopyOwned--------------------------------------------------------------------
 Ξ Library
 c?:Copy
 rep!: Report
--------------------------------------------------------
 c? ∈ available Y checked-out
 rep! = 'This copy is owned by the library'
---------------------------------------------------------------------------------------------
```

Thus, the total specification of the transaction of adding a book entirely new to the library is given by the schema *DoAddNewBook*.

```
DoAddNewBook ≅ AuthorisedRequestor ∧ AddNewBook ∧ Success
                    ∨ UnauthorisedRequestor
                    ∨ Unregistered
                    ∨ NotNewBook
                    ∨ CopyOwned
```

Adding Another Copy of a Book

```
-----------AddAnotherCopy-------------------------------------------------------
 Δ LibraryState
 c?: Copy
 b?: Book
----------------------------------------------------------
 c? ∉ available Y checked-out
 b? ∈ ran instanceof

 available' = available Y { c? }
 instanceof' = instanceof Y { c? → b? }

 checked-out' = checked-out
 previouslyborrowedby' = previouslyborrowedby
 borrowedby' = borrowedby
 borrower' = borrower
 staff' = staff
 writtenby' = writtenby
 about' = about
---------------------------------------------------------------------------------------------
```

The schema *BookNewToLibrary* and the schema *CopyOwned* capture what happens when the preconditions of the operation *AddAnotherCopy* are violated.

```
-------------------BookNewToLibrary---------------------------------------------
 Ξ Library
 b?: Book
 rep!: Report
--------------------------------------------
 b? ∉ ran instanceof
 rep! = 'This book is new to the Library'
--------------------------------------------------------------------------------
```

The total specification of adding a copy of a book to the library is given by the schema *DoAddAnotherCopy* as follows.

$$\begin{aligned} DoAddAnotherCopy \cong\ & AuthorisedRequestor \wedge AddAnotherCopy \wedge Success \\ & \vee\ UnauthorisedRequestor \\ & \vee\ BookNewToLibrary \\ & \vee\ CopyOwned \end{aligned}$$

4) Removing a Copy of a Book from the Library

The operation of removing a copy of a book from the library is split into two cases: removing a copy of a book while leaving other copies of the same book in the library and removing the only copy of a book from the library.

Removing One of Several Copies

The precondition $c? \in available$ states that we cannot remove a copy that is currently checked out. The predicate $\# (instanceof -1 (| \{ instanceof (c?) \} |)) > 1$ states that there is more than one copy of the book.

The operation *RemoveOther* can go wrong in the following cases: $c? \in checked\text{-}out$, $c? \notin available \wedge c? \notin checked\text{-}out$ and $\#(instanceof -1 (| \{ instanceof (c?) \} |)) = 1$. $c? \in checked\text{-}out$ is captured by *CopyCheckedOut*, $c? \notin available \wedge c? \notin checked\text{-}$

out is captured by the schema *CopyNotOwned* , and $\#(instanceof-1 (| \{ instanceof(c?) \} |)) = 1$ is captured by the schema *OnlyCopy*.

-------------*RemoveOther*---

$\Delta Library$
$\Xi LibrarySystem$
$c?: Copy$

$c? \in available$
$\# (instanceof^{-1} (| \{ instanceof(c?) \} |)) > 1$

$available' = available \setminus \{ c? \}$
$previouslyborrowedby' = \{ c? \} \mathbin{\lhd\!\!\!-} previouslyborrowedby$

$checked\text{-}out' = checked\text{-}out$
$borrowedby' = borrowedby$
$borrower' = borrower$
$staff' = staff$

-----------*OnlyCopy*--

$\Xi Library$
$c?: Copy$
$rep!: Report$

$\#(instanceof^{-1} (| \{ instanceof(c?) \} |)) = 1$
$rep! = \text{'This is the only copy in the library'}$

Thus, the total specification of the operation of removing one of several copies of a book from the library is given by the schema *DoRemoveOther* as follows.

$$
\begin{aligned}
DoRemoveOther \cong\ & AuthorisedRequestor \wedge RemoveOther \wedge Success \\
& \vee\ UnauthorisedRequestor \\
& \vee\ CopyCheckedOut \\
& \vee\ CopyNotOwned \\
& \vee\ OnlyCopy
\end{aligned}
$$

Removing the Only Copy of a Book

RemoveLast

$$\begin{array}{l}
\Delta\, Library \\
c?: Copy
\end{array}$$

$$\begin{array}{l}
c? \in available \\
\#(\, instanceof^{-1}\, (\mid \{\, instanceof(\, c?\,)\,\} \mid)\,) = 1 \\
\\
available' = available \setminus \{\, c?\,\} \\
previouslyborrowedby' = \{\, c?\,\} ⩤ previouslyborrowedby \\
instanceof' = \{\, c?\,\} ⩤ instanceof \\
writtenby' = \{\, instanceof(\, c?\,)\,\} ⩤ writtenby \\
about' = \{\, instanceof(\, c?\,)\,\} ⩤ about \\
\\
checked\text{-}out' = checked\text{-}out \\
borrowedby' = borrowedby \\
borrower' = borrower \\
staff' = staff
\end{array}$$

The operation *RemoveLast* can go wrong in the following ways: $c? \in checked\text{-}out$, $c? \notin available \wedge c? \notin checked\text{-}out$, and $\#(\, instanceof-1\, (\mid \{\, instanceof(\, c?\,)\,\} \mid)\,) > 1$. The predicate $c? \in checked\text{-}out$ is captured by *BookCheckedOut*, $c? \notin available \wedge c? \notin checked\text{-}out$ is captured by the schema *CopyNotOwned*, and the predicate $\#(\, instanceof-1\, (\mid \{\, instanceof(\, c?\,)\,\} \mid)\,) > 1$ is captured by the schema *NotOnlyCopy*.

NotOnlyCopy

$$\begin{array}{l}
\Xi\, Library \\
c?: Copy \\
rep!: Report
\end{array}$$

$$\begin{array}{l}
\#(\, instanceof^{-1}\, (\mid \{\, instanceof(\, c?\,)\,\} \mid)\,) > 1 \\
rep! = \text{'This is not the only copy in the library'}
\end{array}$$

The total specification is captured by the following schema.

$$\begin{array}{rl}
DoRemoveLast \cong & AuthorisedRequestor \\
& \wedge\ RemoveLast \\
& \wedge\ Success
\end{array}$$

$\vee$ *UnauthorisedRequestor*

$\vee$ *CopyCheckedOut*

$\vee$ *CopyNotOwned*

$\vee$ *NotOnlyCopy*

5) Getting the List of Books by a Particular Author

---*ByAuthor*---

Ξ *Library*

a?: Author

out!: F Book

----

out! $= \{$ *x: Book* $|$ *a?* $\in$ *writtenby x* $\}$

DoByAuthor $\cong$ *ByAuthor* $\wedge$ *Success*

6) Getting the List of Books on a Particular Subject

-----*BySubject*---

Ξ *Library*

s?: Subject

out!: F Book

----

out! $= \{$ *x: Subject* $|$ *s?* $\in$ *about x* $\}$

DoBySubject $\cong$ *BySubject* $\wedge$ *Success*

7) Finding out the List of Books Currently Checked out by a Particular Borrower

-----*BooksBorrowedBy*--

Ξ *Library*

n?: Person

out?: F Copy

----

n? $\in$ *borrower*

out! $=$ *borrowedby* $^{-1}$ $(| \{ n? \} |)$

This transaction can be performed by a staff user for any borrower or by a borrower for himself or herself.

```
----------SelfRequestor-----------------------------------------------------------
| Ξ Library
| n?, requestor?: Person
|---------------------------------------------------
| n? = requestor?
|-----------------------------------------------------------------------------------
```

This transaction is unsuccessful if one tries to get information about someone's borrowings if they are not registered users.

```
------UnknownBorrower------------------------------------------------------------
| Ξ Library
| n?: Person
| rep!: Report
|---------------------------------------------------
| n? ∉ borrower
| rep! = ' Unknown borrower'
|-----------------------------------------------------------------------------------
```

This transaction can go wrong if it is requested neither by a staff member of the library nor by a registered user.

```
-----NotAuthorisedRequestor-----------------------------------------------------
| Ξ Library
| n?, requestor?: Person
| rep!: Report
|------------------------------------------------------
| n? = requestor?
| Requestor? ∉ staff
| Rep! = 'Not authorized requestor'
|-----------------------------------------------------------------------------------
```

The total specification is given by the following schema.

$$\begin{aligned} DoBooksBorrowedBy \cong\ & BooksBorrowedBy \wedge AuthorisedRequestor \wedge Success \\ & \vee BooksBorrowedBy \wedge SelfRequestor \wedge Success \\ & \vee UnknowBorrower \\ & \vee NotAuthorisedRequestor \end{aligned}$$

8) Finding out Who Last Checked out a Particular Copy of a Book

----*PreviousBorrower*----

$\Xi\ Library$

$n!: Person$

$c?: Copy$

$c? \in available$ Y $checked\text{-}out$

$\{ c? \rightarrow n! \} \in previouslyborrowedby$

This transaction fails if either the copy *c?* is not owned by the library or the copy *c?* has never been borrowed.

----*CopyNotPreviouslyBorrower*----

$\Xi\ Library$

$rep!: Report$

$c? \notin dom\ previouslyborrowedby$

$rep! =$ *'Copy not previously borrowed'*

Thus, the total specification of the operation is given as follows.

$$\begin{aligned} DoPreviousBorrower \cong\ & PreviousBorrower \wedge AuthorisedRequestor \wedge Success \\ & \vee UnauthorisedRequestor \\ & \vee CopyNotOwned \\ & \vee CopyNotPreviouslyBorrower \end{aligned}$$

4.2.2.2 Discussion of the Specification in Z

The specification in Z is built from near orthogonal components. Users look for ways of splitting the state of the system so that operations on the required state can be specified.

For the library problem, the specification is split into three parts: entities, the states of the system, and the operations of the system. Global entities provide static type check. The states of the system are defined using schemas and global entities. It is clear that the library system includes the library management system and the library database. The initial states are defined. In the operation part, eight transactions of the library problem are specified one by one. The states were divided into small groups of components and sub-operations in order to define the full operation. For a large specification with many state components, this approach would be clumsy.

The precondition of an operation characterizes exactly these inputs and initial states such that there exists at least one possible combination of outputs and a final state that satisfies the operation specification.

4.2.3 UML Approach

4.2.3.1 Specification in UML

In this section, textual English and UML notations are used to specify and analyze the library system. Identifying use cases and behaviors of the system follow the previously overviewed process in Section 2.2.5.

There are eight transactions, two kinds of users, and three constraints in the library system. The system needs to present the functions with a representation that can be a graphical user interface. The system also needs a persistent storage mechanism, such as an object-oriented or a relational database. Finally, it is important that the system needs an application system that drives this system. Figure 4.1 shows components of the system that interact with each other.

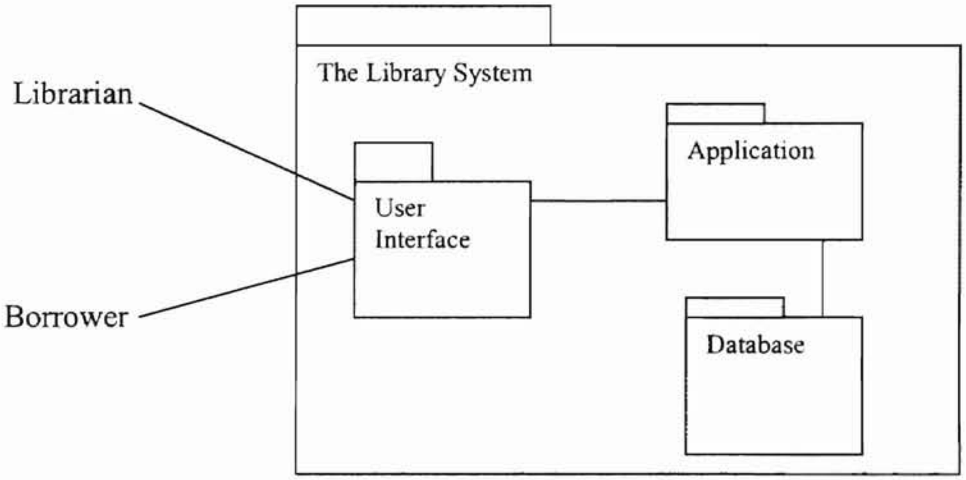


Figure 4.1 The Library System Components

The following specification will focus on the application part in order to specify the requirement and to have a fair comparison with Z.

4.2.3.1 Identifying Use Cases

Identifying use cases includes identifying actors and use cases, drawing use case diagrams, documenting use cases, and representing each use case using sequence diagram scenarios.

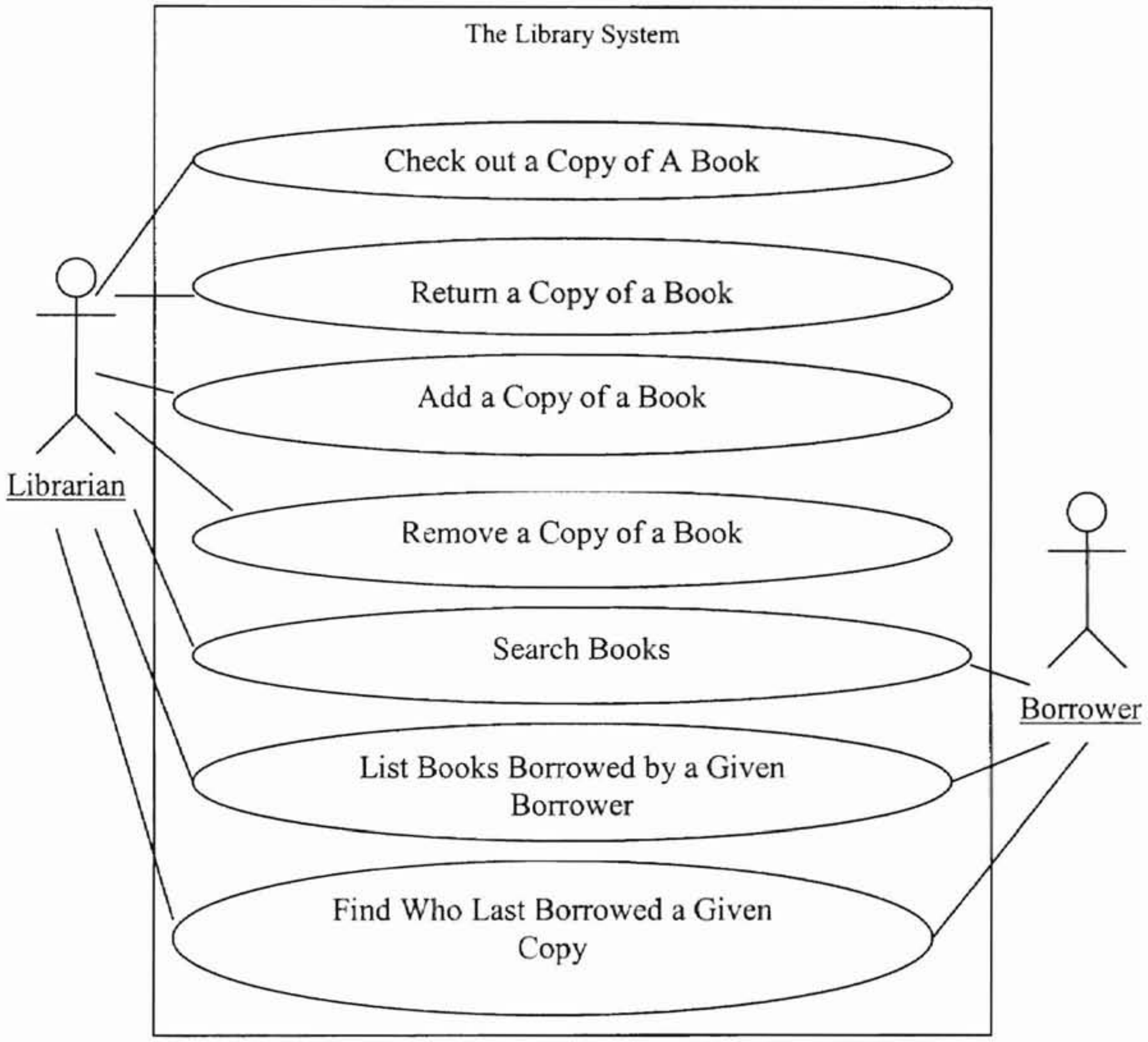


Figure 4.2 Main Use Case Diagram of the Library Problem

In this system, there are two actors: a borrower and a librarian. There are seven use cases that are represented in Figure 4.2. These use cases describe the functionality of the library from the actor point of view. The boundary of this system is the system software in the library.

A librarian refers to any person who is a staff of the library, and responsible to do all transactions. A borrower refers to any person who is legal to borrow books from this library. The following is the documentation of seven use cases.

1) Check out a Copy of a Book

This use case starts when a borrower arrives at the system with a copy of a book to check out. A librarian shows he/she is a valid librarian with an identification card. The system gives him/her permission to do this transaction. The librarian makes sure that the borrower is a valid borrower and that the book is available, makes sure that the borrower has not borrowed more than the maximum number of copies, records this copy on the borrower's list, and finally records that this book has been borrowed. This use case ends.

Alternative courses: The librarian cannot show he/she is a valid librarian, the system stops this transaction. The borrower cannot show she/he is a valid borrower, the system stops this transaction. The book is not available, the system stops this transaction.

The sequence diagram scenario is given in Figure 4.3.

2) Return a Copy of a Book

This use case starts when a borrower arrives at the system with a copy of a book to return. A librarian shows he/she is a valid librarian with an identification card, and the system gives him/her permission to do this transaction. The librarian makes sure that the borrower is a valid borrower, this copy belongs to the library, and is now available to the library. Then the librarian records this copy as available to the library and removes this copy from the borrower's list. This use case ends.

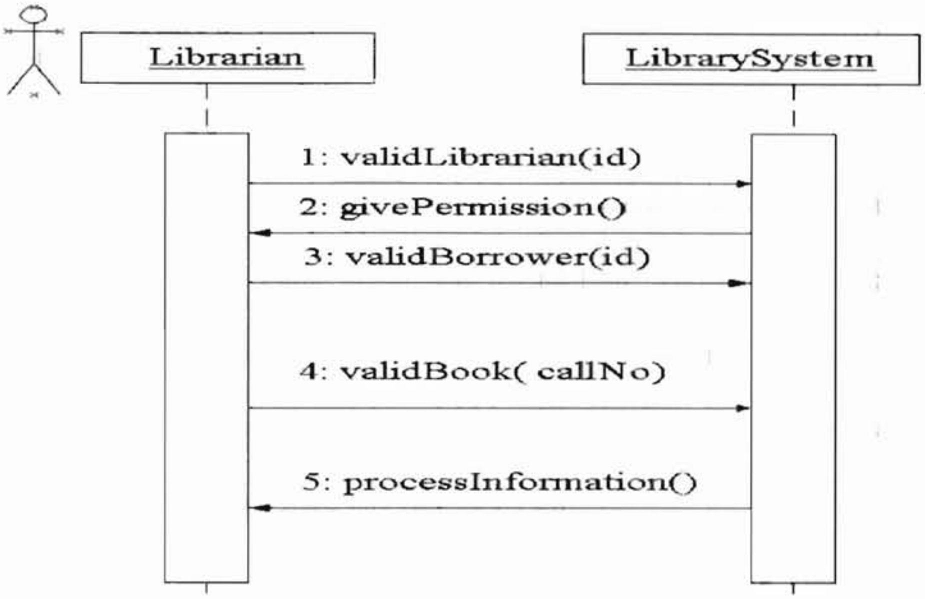


Figure 4.3 Sequence Diagram Scenario of Check out a Copy of a Book

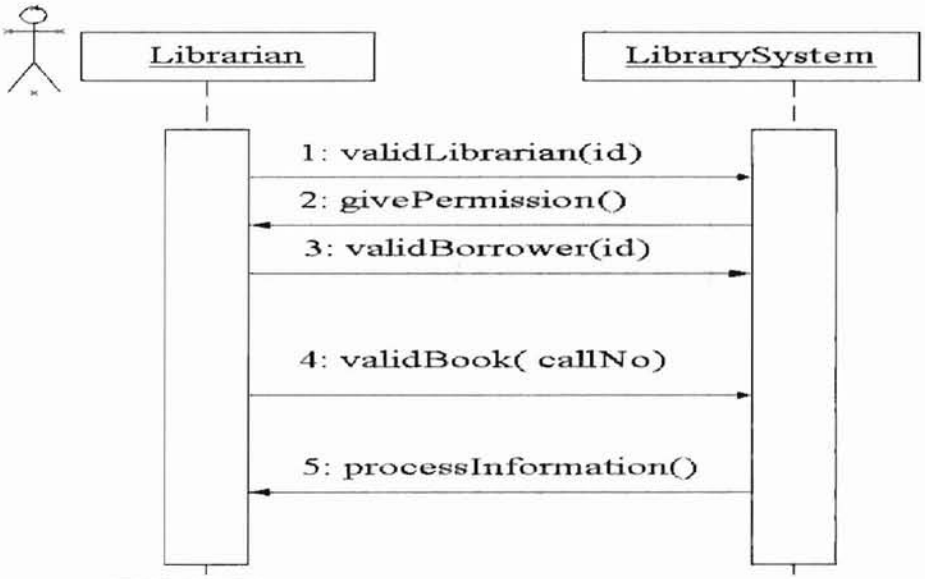


Figure 4.4 Sequence Diagram Scenario of Return a Copy of a Book

Alternative courses: The librarian cannot show he/she is a valid librarian, the system stops this transaction. The borrower cannot show she/he is a valid borrower, this transaction is stopped. The book is not available, the system stops this transaction. The sequence diagram scenario is given in Figure 4.4.

3) Add a Copy of a Book

This use case starts when a librarian shows he/she is a valid librarian and the system gives him/her permission to do this transaction. The system checks the book status. If the book is new to the library, the system initiates Add a New Book use case, otherwise the system initiates Add Another Copy of a Book use case. This use case ends. Figure 4.5 represents the subsystem use case diagram.

Alternative courses: The librarian cannot show he/she is a valid librarian, this transaction is stopped. The sequence diagram scenario is given in Figure 4.6.

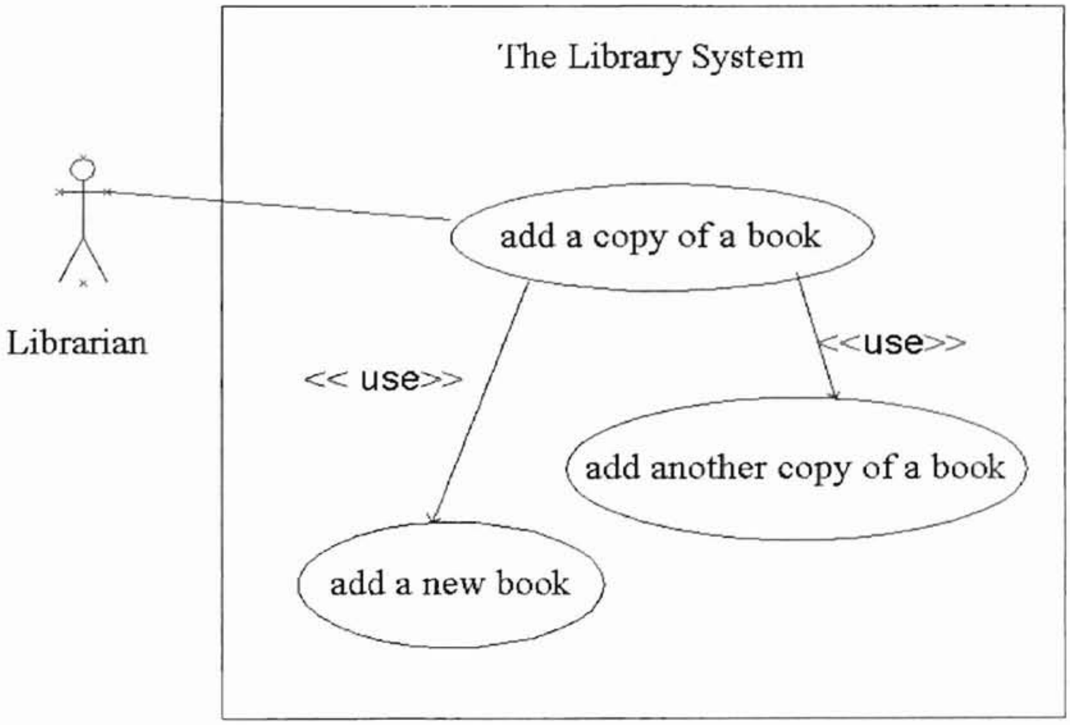


Figure 4.5 Subsystem Use Case Diagram of Add a Copy of a Book

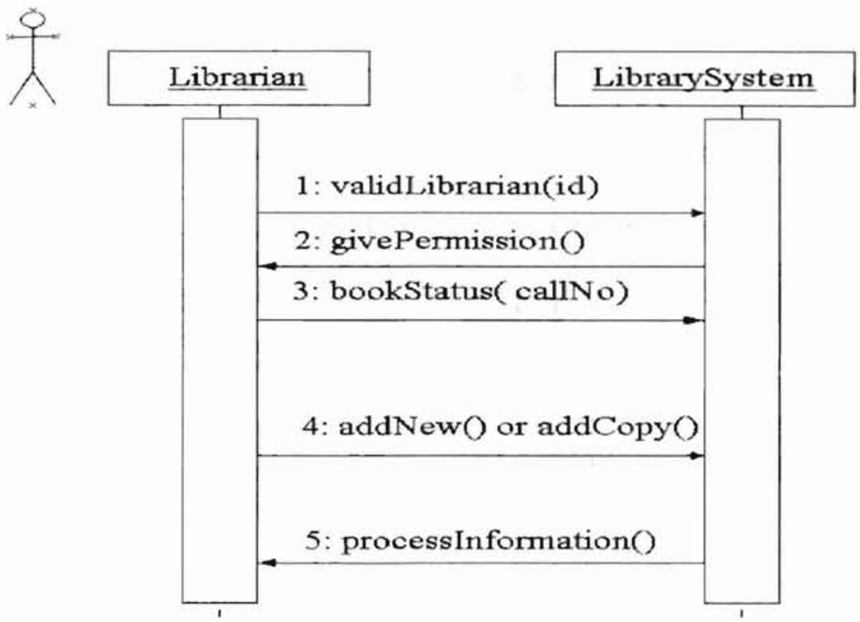


Figure 4.6 Sequence Diagram Scenario of Add a Copy of a Book

The Subsystem Use Cases of Add a Copy of a Book is given below.

Add a new book

This use case starts when the system has verified that the book is new to the library. The librarian adds this book as one copy in the library database, adds the author and subject to the book, and finally adds it as available to the library. This use case ends.

Add Another Copy of a Book

This use case starts when the system has verified that the book is not new to the library. The system finds the book and increases the number of copies. The system finally adds it as available in the library. This use case ends.

4) Remove a Copy of a Book

This use case starts when a librarian shows he/she is a valid librarian, and the system gives him/her permission to do this transaction. The system finds how many copies the book has. If there is only one copy of the book, the system initiates Remove the Only Copy use case. Otherwise, the system initiates Remove Another Copy use case. This use case ends. Figure 4.7 represents the subsystem use case diagram.

Alternative courses: The librarian cannot show he/she is a valid librarian, this transaction is stopped. The sequence diagram scenario is given in Figure 4.8.

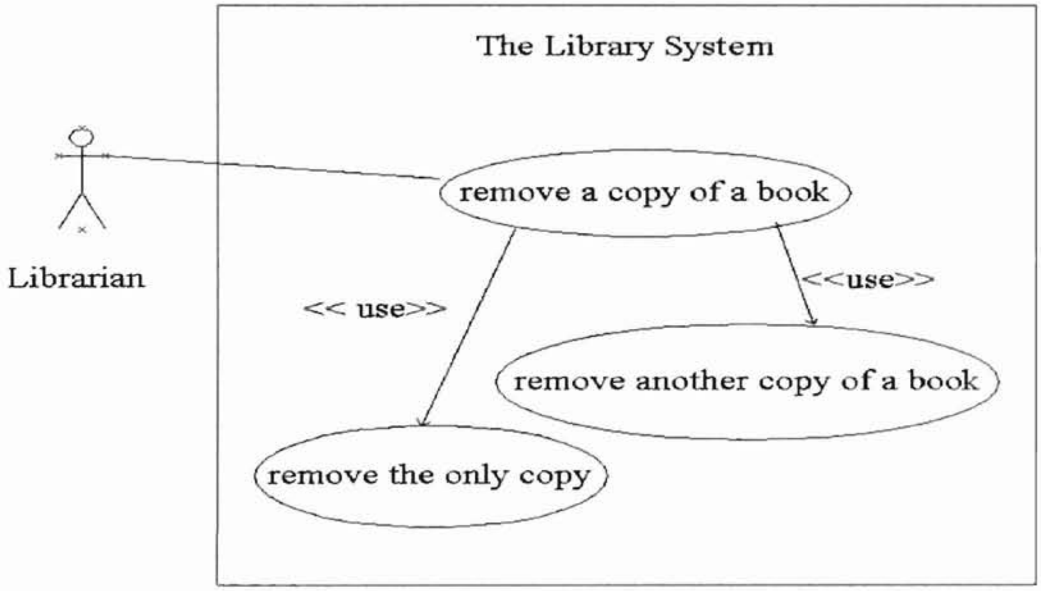


Figure 4.7 Subsystem Use Case Diagram of Remove a Copy of a Book

The Subsystem Use Cases of Remove a Copy of a Book is given below.

Remove the Only Copy of a Book

This use case starts when the system finds that this book has only one copy. The system removes the copy from the available copies of the book, removes the author and subject from the book, and removes the book from the database. This use case ends.

Remove Another Copy of a Book

This use case starts when the system has showed there are more than one copy of a book. The system removes the copy from the available list. This use case ends.

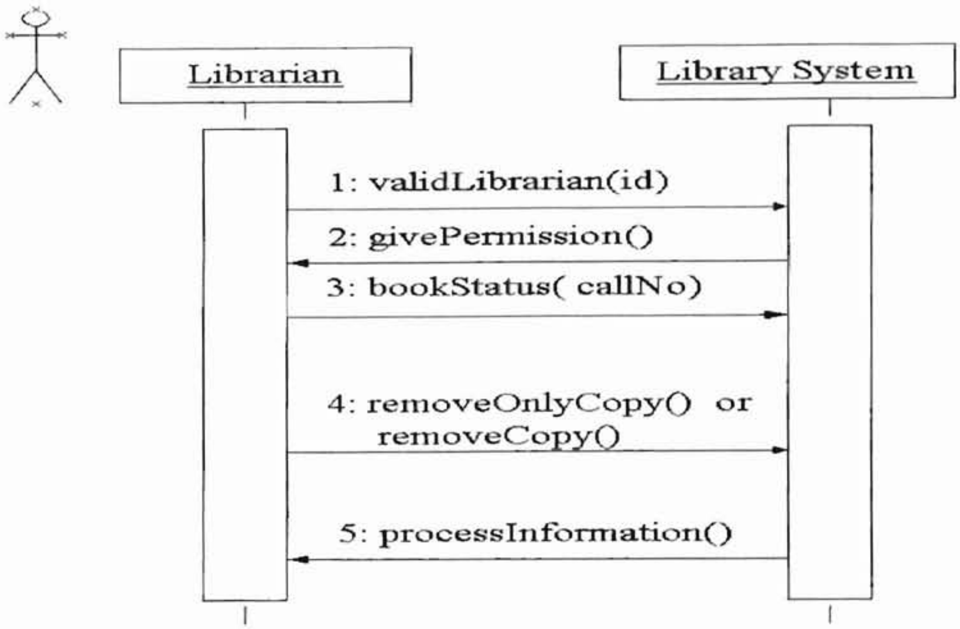


Figure 4.8 Sequence Diagram Scenario of Remove a Copy of a Book

5) Search Books

This use case starts when a user wants to search for a specific book. The user can choose the author as the key or the subject as the key. The system outputs a list of books that match the key. This use case ends. The sequence diagram scenario is given in Figure 4.9.

6) List Books Borrowed by a Given Borrower

This use case starts when a user wants to list books borrowed by a given borrower. The system requires the user to show his/her status. If the user is a librarian, the system initiates List by a Librarian use case. If the user is a borrower, the system initiates List by a Borrower use case. Otherwise the system stops this transaction. This use case ends. Figure 4.10 represents the subsystem use case diagram.

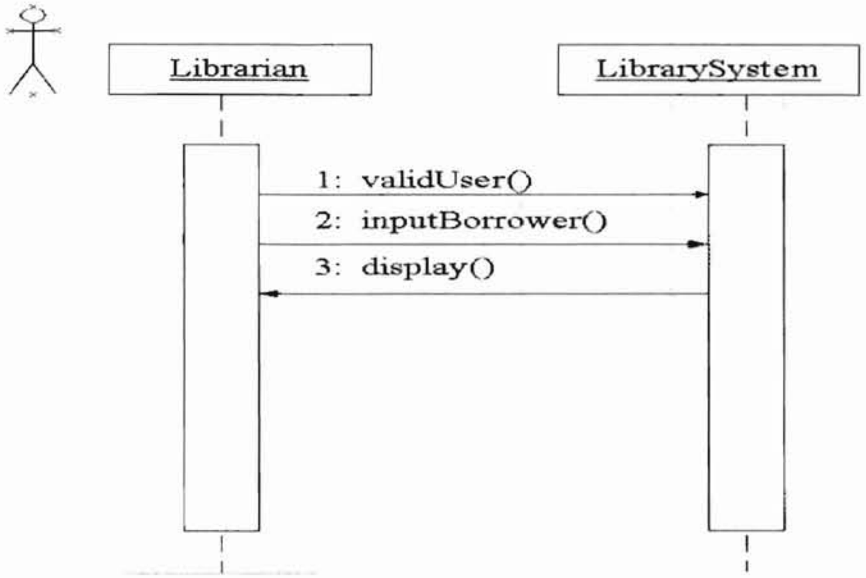


Figure 4.9 Sequence Diagram Scenario of Search Books

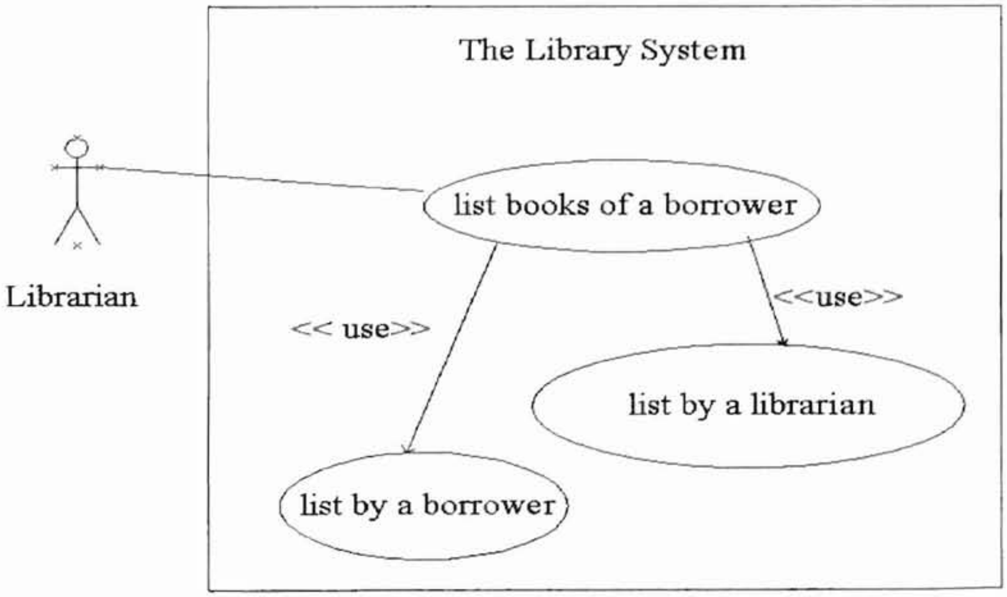


Figure 4.10 Subsystem Use Case Diagram of List Books Borrowed by a Given Borrower

The sequence diagram scenario for the use case of List Books Borrowed by a Given Borrower is in Figure 4.11.

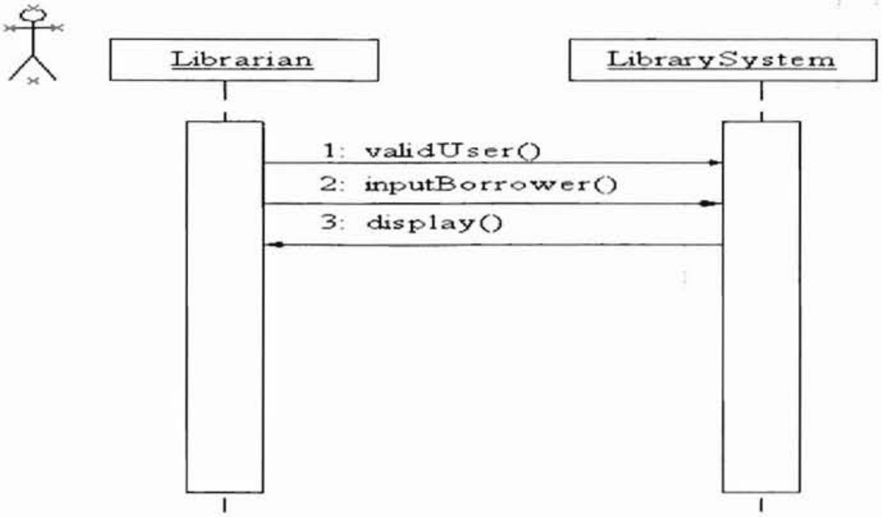


Figure 4.11 Sequence Diagram Scenario of List Books Borrowed by a Given Borrower

The Subsystem Use Cases of List Books Borrowed by a Given Borrower is given as below.

List by a Librarian

This use case starts when the system finds the user is a librarian. The librarian inputs the borrower's id and the system outputs a list of books borrowed by the borrower. This use case ends.

List by a Borrower

This use case starts when the system finds the user is a borrower. The system outputs a list of books borrowed by the borrower. This use case ends.

7) Find out Who Last Borrowed a Given Copy

This use case starts when a user shows he/she is a librarian. The librarian inputs the copy number and the system outputs the last borrower or outputs a message that the copy is not owned by the library. This use case ends. The sequence diagram scenario is given in Figure 4.12.

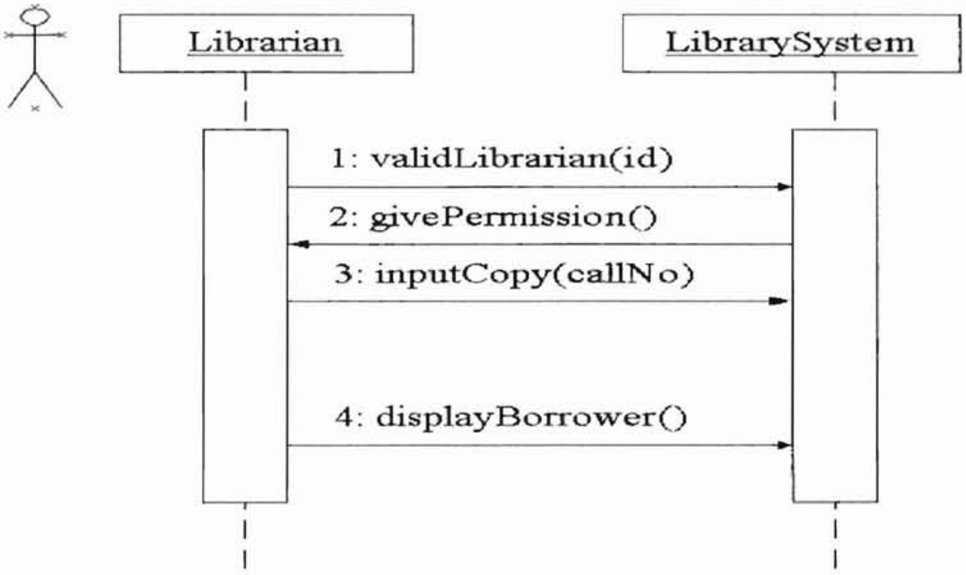


Figure 4.12 Sequence Diagram Scenario of Find out Who Last Borrowed a Given Copy

4.2.3.1.2 Identifying Behaviors of the System

In this section, a collaboration diagram and a state diagram for each use case are drawn to show the behaviors of the use cases. The collaboration diagrams describe the collaboration of objects to realize the functions of the system. Only the main methods are showed to simplify the diagrams. The arrow represents the direction. The state diagrams describe the different states of the systems.

1) Check Out a Copy of a Book

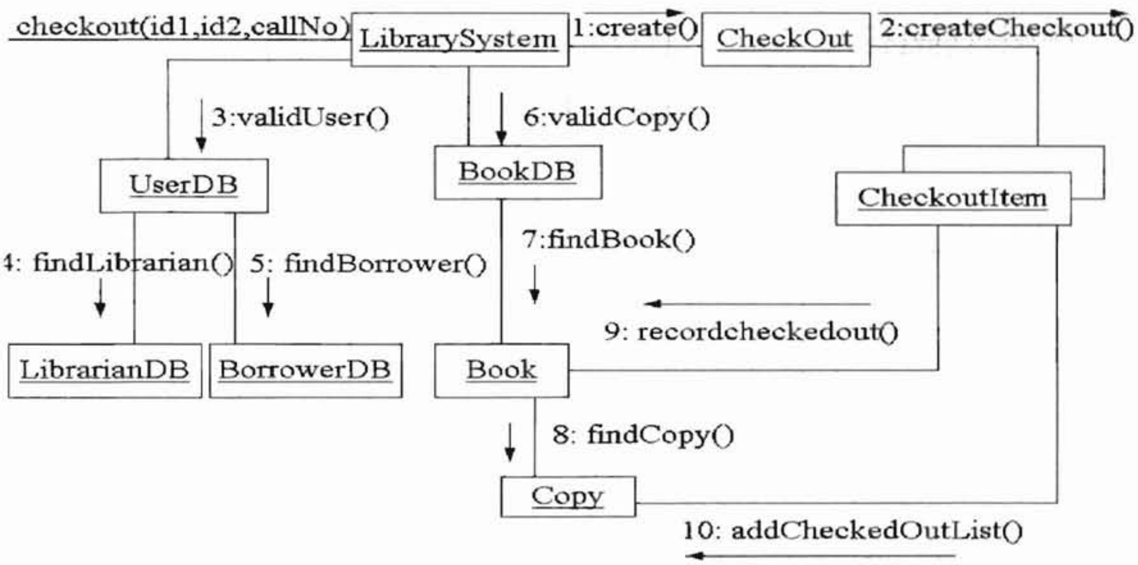


Figure 4.13 Collaboration Diagram of Check out a Copy of a Book

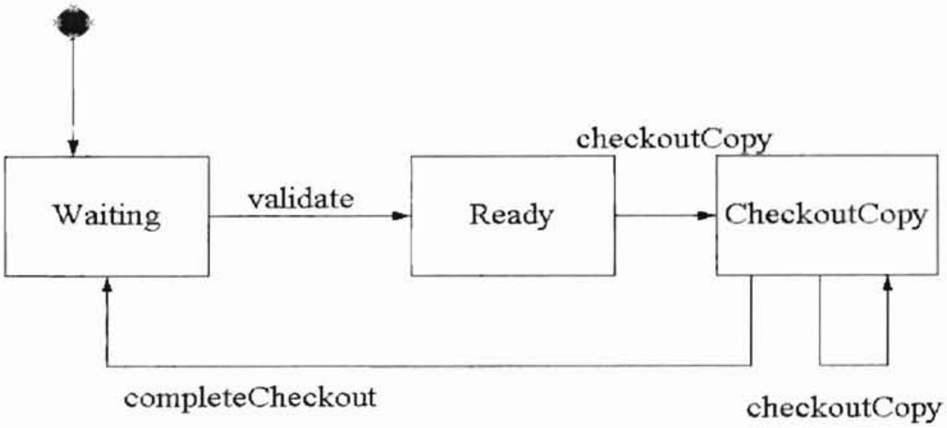


Figure 4.14 State Diagram of Check out a Copy of a Book

2) Return a Copy of a Book

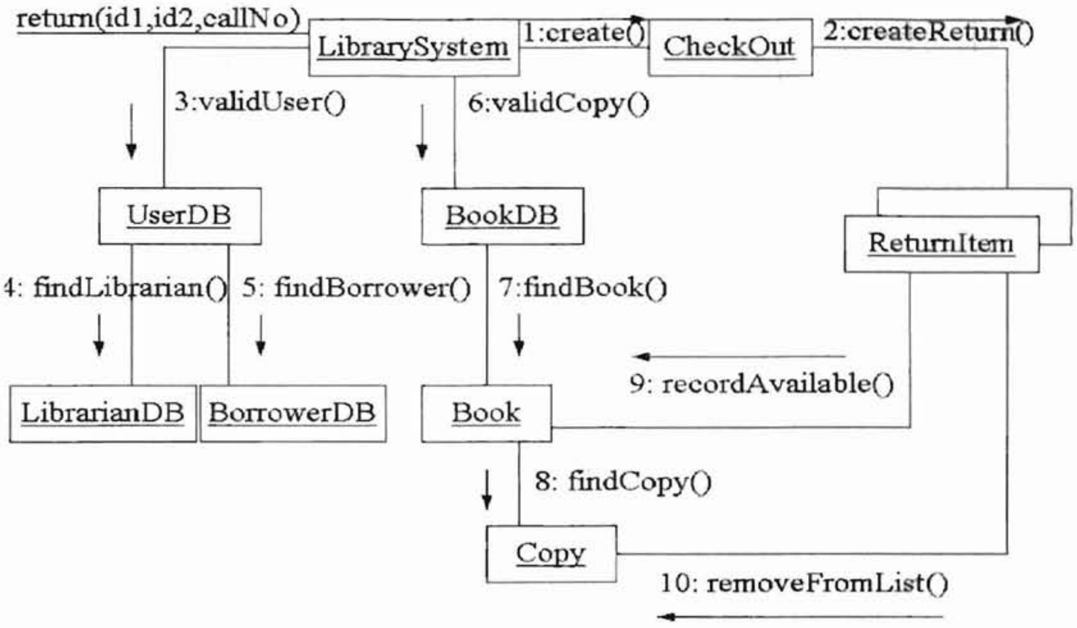


Figure 4.15 Collaboration Diagram of Return a Copy of a Book

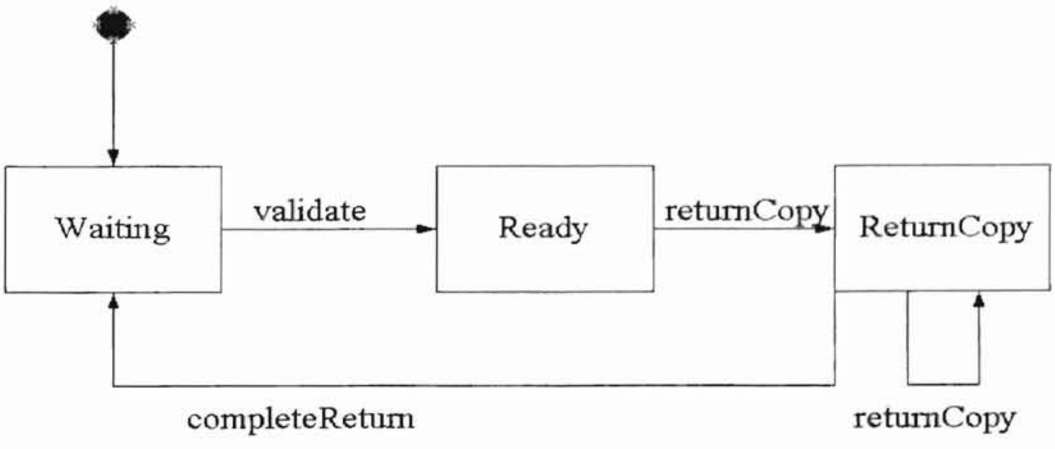


Figure 4.16 State Diagram of Return a Copy of a Book

3) Add a Copy of a Book

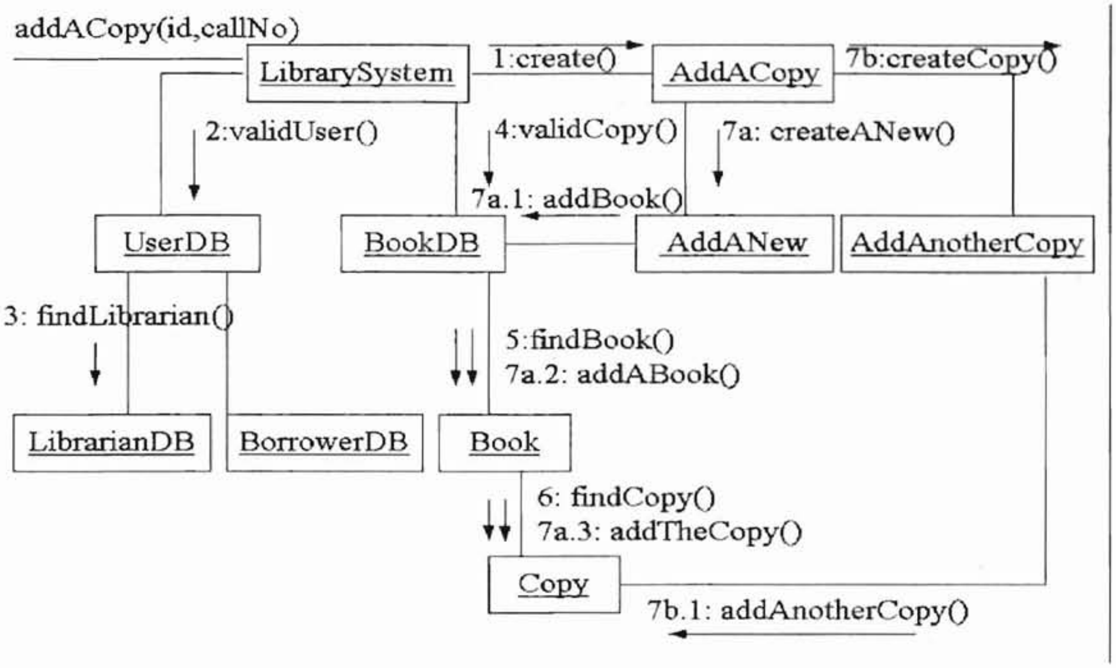


Figure 4.17 Collaboration Diagram of Add a Copy of a Book

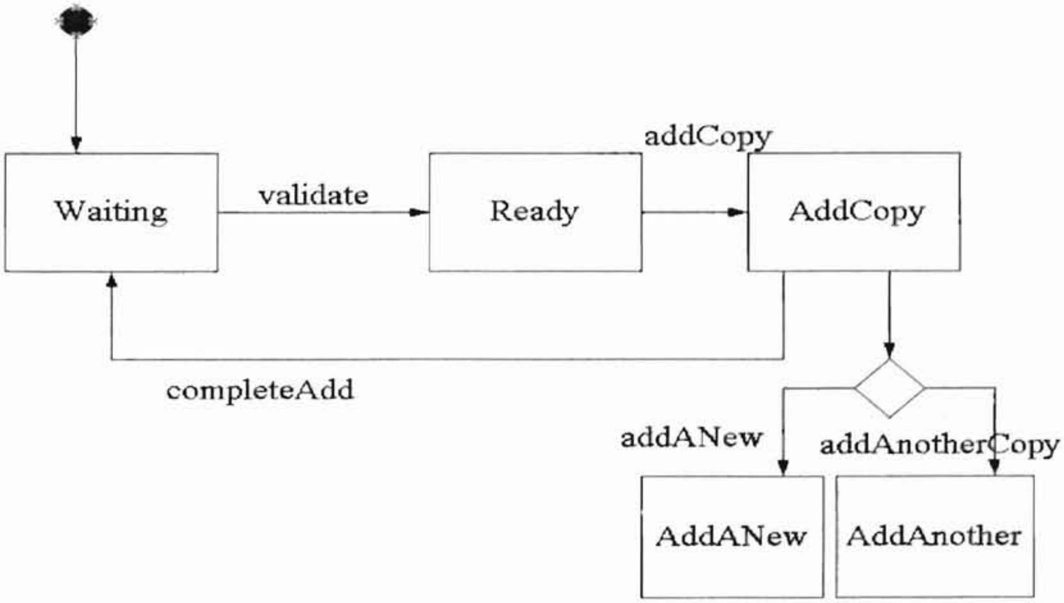


Figure 4.18 State Diagram of Add a Copy of a Book

4) Remove a Copy of a Book

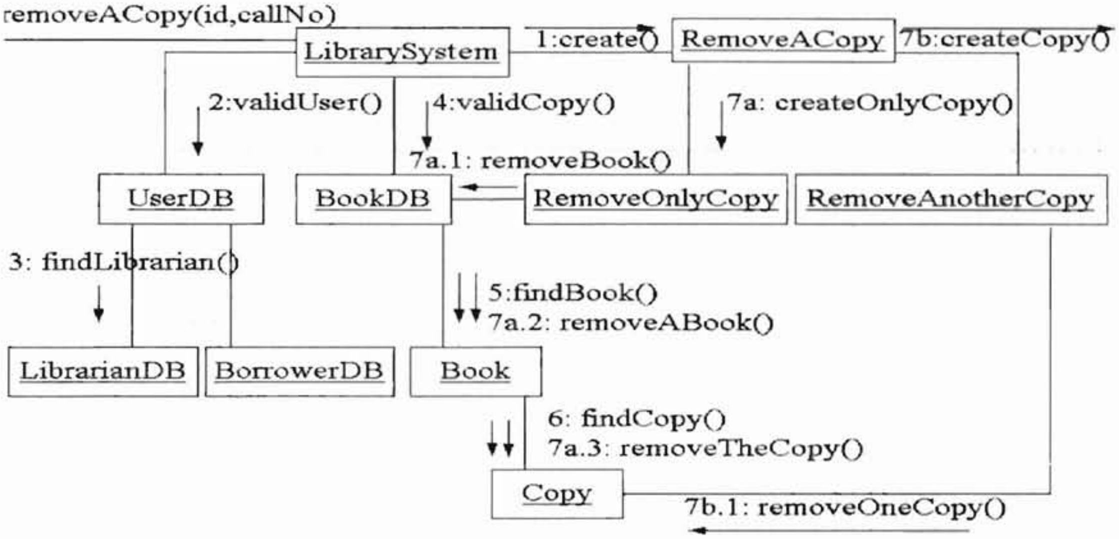


Figure 4.19 Collaboration Diagram of Remove a Copy of a Book

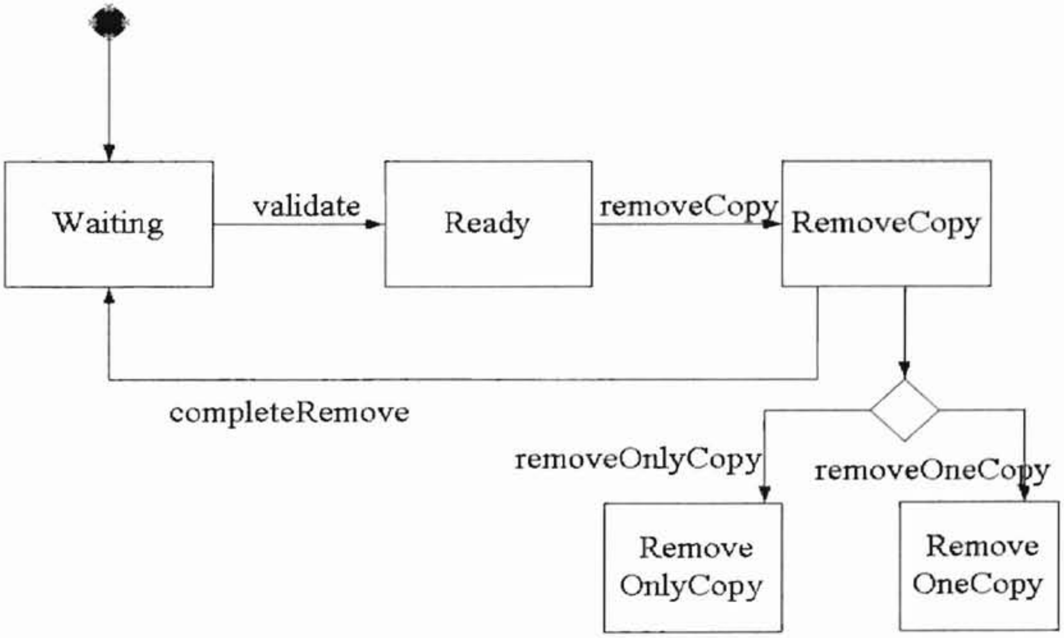


Figure 4.20 State Diagram of Remove a Copy of a Book

5) Search Books

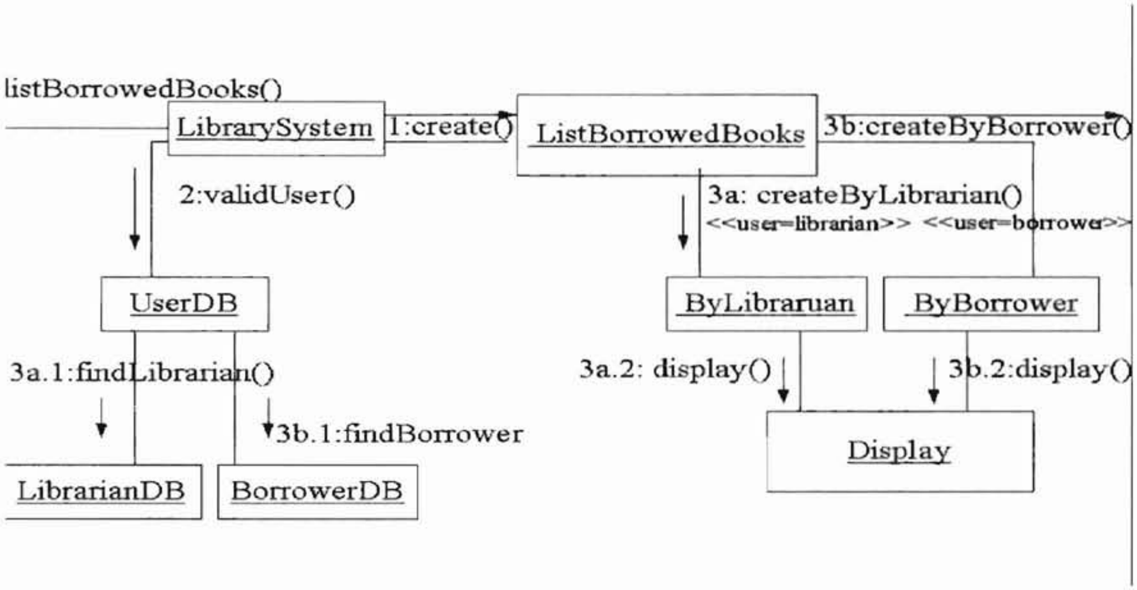


Figure 4.21 Collaboration Diagram of Search Books

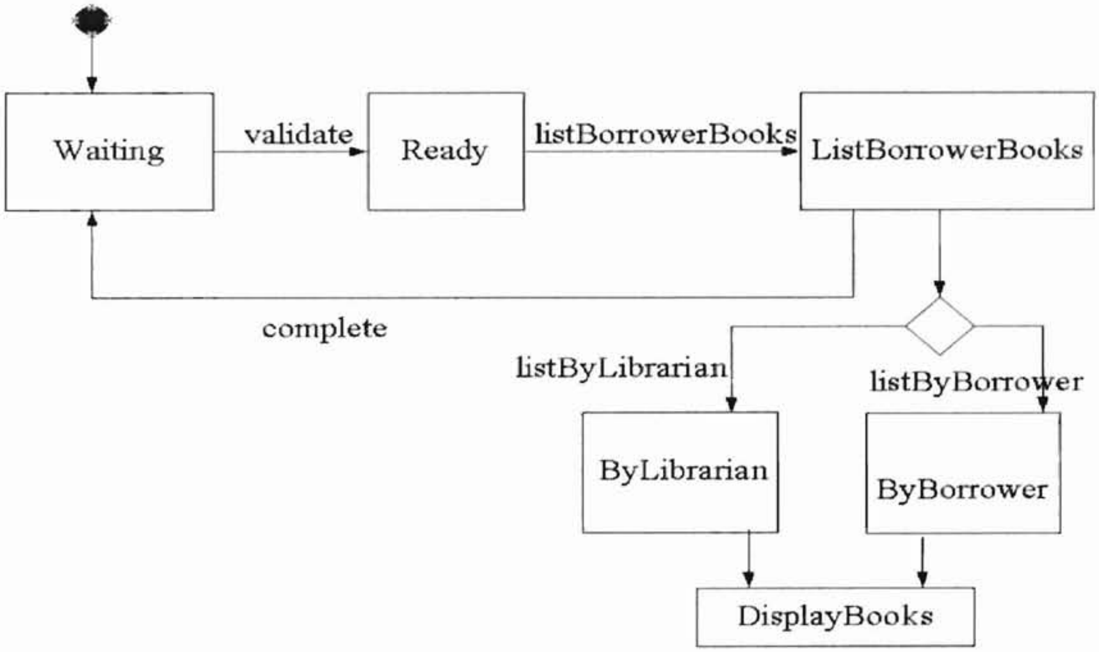


Figure 4.22 State Diagram of Search Books

6) List Books Borrowed by a Given Borrower

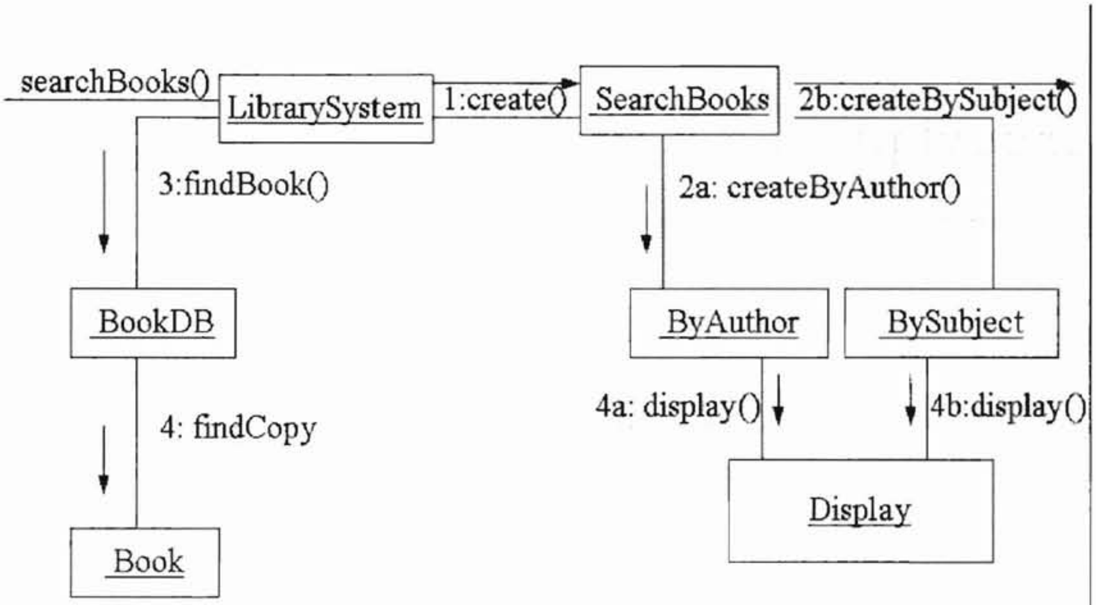


Figure 4.23 Collaboration Diagram of List Books Borrowed by a Given Borrower

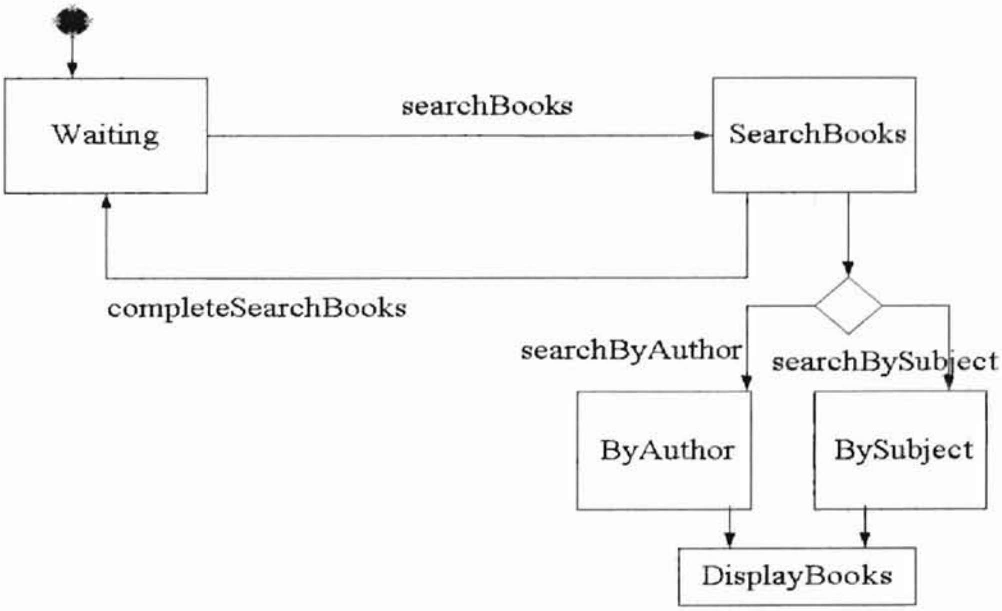


Figure 4.24 State Diagram of List Books Borrowed by a Given Borrower

7) Find Out Who Last Borrowed a Given Copy

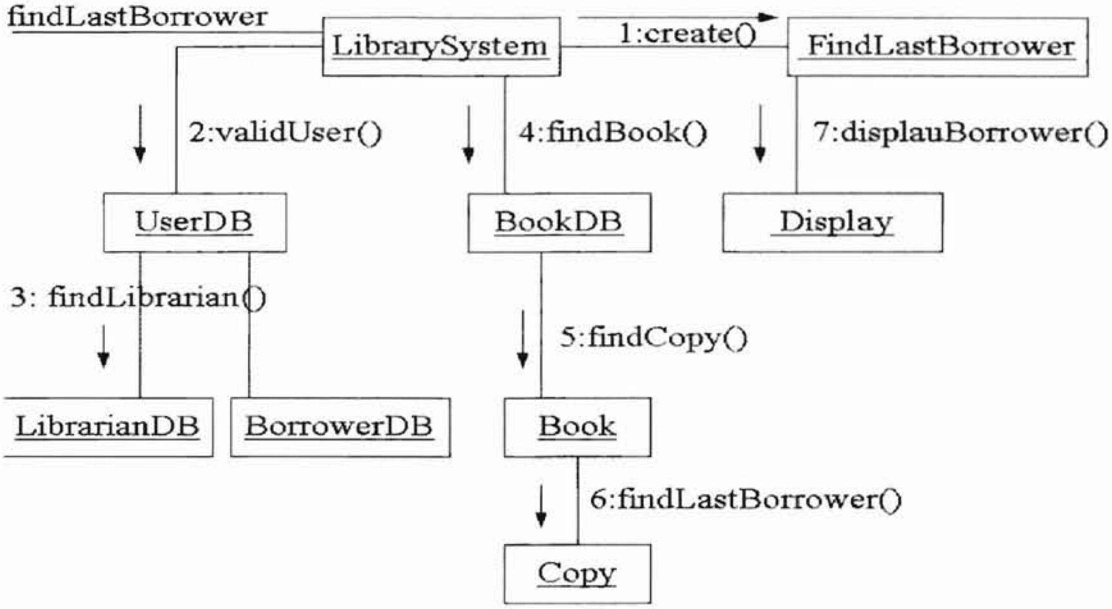


Figure 4.25 Collaboration Diagram of Find Out Who Last Borrowed a Given Copy

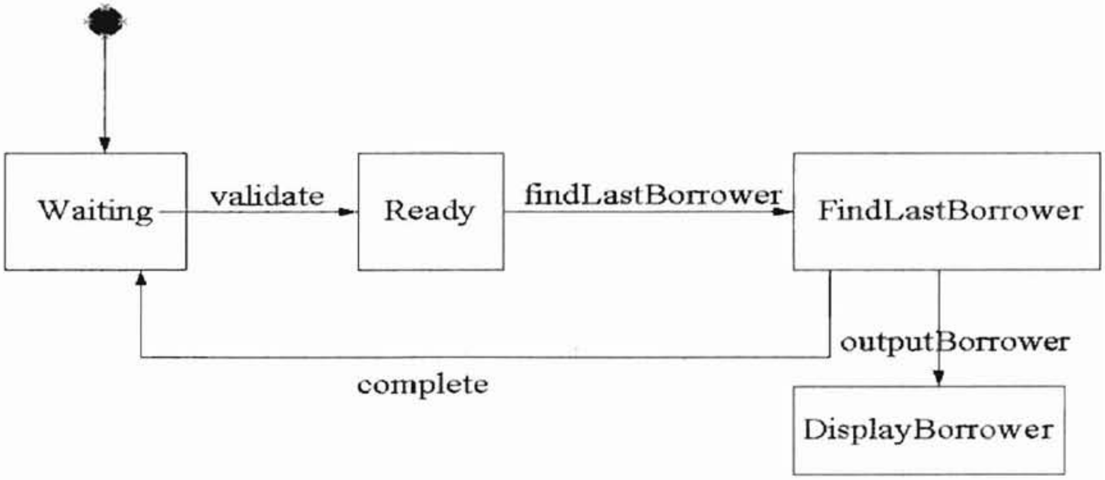


Figure 4.26 State Diagram of Find Out Who Last Borrowed a Given Copy

4.2.3.2 Discussion of the Specification in UML

The use cases and use case diagrams illustrate the basic requirements of the library system, and how the system will be used.

The collaboration diagrams describe the sequence and collaboration between objects. The state diagrams describe the transition of state or activities of the use case. These are easy to understand. But in contrast to the specification in Z, they are neither precise nor concise.

4.2.4 Comparison of Z and UML Approaches

Z and UML are compared based on the criteria identified in Section 4.1.

4.2.4.1 Readability

The specification in Z uses semi-graphical schemas and mathematical symbols. The textual English style makes it hard for the reader to understand the entire system. In contrast, the specification in UML uses graphical notations and meta-model diagrams. The library use case diagram and the sequence diagram scenario for each use case specify the problem requirements well. The collaboration diagrams and state diagrams describe the entire system and, the related relationships clearly.

4.2.4.2 Applicability

Both specifications in Z and UML are expressed in a natural and straightforward manner. This means both Z and UML are applicable to specify the library problem.

4.2.4.3 Levels of Abstract / Expressibility

The specification in Z is in a lower abstract level. Entities, states, and operations are used to specify the library problem. This specification does not specify the external environment. The specification in Z is quite detailed about the system. So, a lot of textual English sentences are used to explain the abstraction structuring.

In contrast with the specification in Z, the specification in UML is at a higher abstract level. Classes are the basic building blocks. Meta-model diagrams represent the entire system and the relationships. A few English sentences are used to explain the specification. The external environment is specified using actors.

4.2.4.4 Verifiability

The specification in Z can be verified using first-order predicate logic and set theory. Does this specification meet the requirements of the library problem? The best way is to implement it and test it. But we find this specification is far away from implementation. In contract, the specification in UML is closer to the implementation. But the specification cannot be proved to be correct.

4.2.4.5 Discipline

The specification in Z forces users to write reasonably well-behaved specifications. Users have to use mathematical notations and strict first-order predicate logic to explicitly and implicitly describe the preconditions and the post-conditions.

The specification in UML does not necessarily help users to write modular specifications. Users just use the notation and the diagram to describe the system.

4.2.4.6 Reusability

Z and UML both have mechanisms to reuse parts of the respective model specifications in formalizing other models. Specifications in Z use the symbol Ξ to represent the specification operations that do not change the state and use the symbol Δ to represent the specification operations that change the state. Specifications in UML use classes to represent a set of objects and aggregates to indicate the composite of whole and part relationships.

4.2.6.7 Ambiguity

A set of questions is posted to help evaluate the ambiguity of specifications in Z and UML. Table 4.1 summarizes how each specification treats the five major ambiguity areas.

Table 4.1 Ambiguities of Specifications in Z and UML

Specifications	Z	UML
Library	Specified	Not specified
User	Specified	Not specified
Book	Specified	Specified
Available	Specified	Not specified
Last Checked out	Specified	Not specified

The five questions are as follows.

1. What is the difference between the library system and the library database? The specification in Z clearly distinguishes a library database from the entire library system. The specification in UML fails.

2. What is a user? The specification in Z and UML both declare that there are two kinds of user: librarians and borrowers. The specification in Z declares.

{ librarian } I { borrower } = { }.

The specification in UML does not specify it.

3. What is a book? Both specifications in Z and UML declare a book and a copy of a book as different entities.

4. What does "available" means? The specification in Z clearly distinguished between "available" and "checked out". The specification in UML fails.

5. What does "last checked out" mean? The specification in Z clearly distinguished between "last" and "current", but the specification in UML fails.

From the above comparison, it is clear that Z is more precise than UML.

4.3 Elevator Control Problem Case Study

An introduction to the problem is given in Section 4.3.1. The Z approach and the UML approach are given in Sections 4.3.2 and 4.3.3, respectively. A comparison of these specifications based on the criteria given previously (see Section 4.1) is provided in Section 4.3.4.

4.3.1 Introduction to the Problem

The elevator control problem, like the library problem, is a classic problem of software specification. It first appeared in Don Knuth's book in 1968 (as cited in [Schach 97]).

The problem attained broader prominence in 1986 when it appeared in *The Fourth International Workshop on Software Specification and Design* as one of a set of four problems [Schach 97].

This problem has been used as a case study in several references [Schach 97]. This problem is used in this thesis as a case study. The problem was specified by Evans [Evans 94] as follows.

> A lift controller system is required to control a single lift in a multi-story building. Requests for the lift can either be made by customers waiting on different floors of the building, or by customers traveling in the lift. The task of the controller is to service these requests by moving the lift to the required floors. The lift must only service requests in the direction it is currently traveling in. When there are no longer any pending requests in this direction, it will change direction to service any requests in the opposite direction.

4.3.2 Z Approach

4.3.2.1 Specification in Z

This specification has been modified from the one given by Evans [Evans 94]. Textual English and Z notations are mixed in order to explain the problem. The development process follows the previously overviewed process in Section 2.1.5. Global entities, the states of the system, and the operations of the system are the three main components.

4.3.2.1.1 Global Entities

This specification makes use of the following given types: *[MultiFloors, FloorNumber, Direction, State, Door, RequestType]* as explained below. *MultiFloors* is a constant which stands for the number of floors of the building. *FloorNumber*= = 1 … MultiFloors it is a type representing the possible floors used by the elevator.

Direction : : = up | down
State : : = stopped | moving
Door : : = open | closed
RequestType: : = upRequest | downRequest | passengerRequest

4.3.2.1.2 States of the System

The elevator problem is split into *Elevator*, *Request*, *ElevatorSystem*, and *InitializeElevatorSystem* states. Each state is specified below.

```
┌-----Elevator-------------------------------------------------------------------
| position: FloorNumber
| direction: Direction
| state: State
| door: Door
└--------------------------------------------------------------------------------
```

```
┌----Request---------------------------------------------------------------------
| type: RequestType
| floor: FloorNumber
└--------------------------------------------------------------------------------
```

```
------ElevatorSystem-------------------------------------------------------------
| elevator: Elevator
| requests: P Request
---------------------------------------------------------------------------------
┌----InitializeElevatorSystem----------------------------------------------------
| ElevatorSystem
|-------------------------------------------------------------
| elevator'.position = 1 ∧ elevator'.direction = up
| elevator'.state= stopped
| elevator'.door = open
| request' = { }
---------------------------------------------------------------------------------
```

4.3.2.1.3 Operations of the System

There are five operations in this system. They are: *MakeRequest*, *MoveElevatorUp, OpenDoor, CloseDoor,* and *MoveElevatorDown.* Each Operations is specified below.

```
------MakeRequest--------------------------------------------------------------
 $\Delta\, ElevatorSystem$
 $r?: F_1\ Request$
---------------------------------------------------------------
 $requests' = requests\ Y\, r?$
 $left' = lift$
-------------------------------------------------------------------------------------
```

At any moment in time, a number of requests may be made for the elevator. This is specified by the operation *MakeRequest*, which adds the new requests to the set of requests currently pending in the elevator system. It is assumed that this operation is deterministically selected by the users of the elevator. This operation illustrates how two independent components of the concurrent system can communicate with each other via the shared variable requests.

```
-----------MoveElevatorUp--------------------------------------------------------
 $\Delta\, ElevatorSystem$
-------------------------------------------------------------------------------
 $elevator.door = closed$
 $elevator.position < MultiFloors$
 $\neg (\ \exists\ r: Requests \bullet r.floor = elevator.position\ )$
 $\exists\ r: Requests \bullet r.floor > elevator.position\ \wedge$
              $(\ r.type = passengerRequest\ \vee\ r.type = upRequest\ )$
 $(\ elevator.direction = up\ \vee\ (elevator.direction = down\ \wedge\ \neg (\ \exists\ r: Requests \bullet$
       $r.floor < elevator.position\ \wedge\ (r.type = passengerRequest\ \vee$
       $r.type = downRequest\ )\ )\ )\ )$

 $elevator'.position = elevator.position + 1$
 $elevator'.direction = up$
 $elevator'.state = moving$
 $elevator'.door = closed$
 $requests' = requests$
-------------------------------------------------------------------------------------
```

The *MoveElevatorUp* describes this situation: there are pending requests for the elevator above its current position and the elevator is currently travelling upward, or the elevator is travelling downward and there are no remaining downward requests, then this operation will move the elevator up by one floor. This operation will only proceed if there is no request to be serviced at the current floor and the elevator door is closed.

```
|--------OpenDoor--------------------------------------------------------------------
| Δ ElevatorSystem
|-------------------------------------------------------------------
| elevator.door = closed
| ∃ r: Requests • r.floor = elevator.position
|
| elevator'.position = elevator.position
| elevator'.direction = elevator.direction
| elevator'.state = stopped
| elevator'.door = open
| requests' = requests
|------------------------------------------------------------------------------------
```

This operation opens the elevator door when there are floor requests to be serviced and the elevator door is closed.

```
|----------------CloseDoor-------------------------------------------------------------
| Δ ElevatorSystem
|--------------------------------------------------------------------------------
| elevator.door = open
| elevator.state = stopped
| ∃ r: Requests • r.floor ≠ elevator.position
|
| requests' = requests \ { r: Requests | r.floor = elevator.position }
| elevator'.position = elevator.position
| elevator'.direction = elevator.direction
| elevator'.state = moving
| elevator'.door = closed
|--------------------------------------------------------------------------------------
```

This operation closes the elevator door if there are requests at other floors that need service and the elevator door is open. Existing requests that have been serviced by the elevator are now removed from requests.

MoveElevatorDown describes the following situation: there are pending requests for the elevator below its current position and the elevator is currently travelling downward, or the elevator is travelling upward and there are no remaining upward requests, then this operation will move the elevator down by one floor. This operation will only proceed if there is no request to be serviced at the current floor and the elevator door is closed.

```
-----MoveElevatorDown-----------------------------------------------------------
 Δ ElevatorSystem
---------------------------------------------------------------
 elevator.door = closed
 elevator.position >1
 ¬( ∃ r: Requests • r.floor = elevator.position )
 ∃ r: Requests • r.floor < elevator.position ∧
              ( r.type =passengerRequest ∨ r.type = downRequest )
 ( elevator.direction = down ∨ (elevator.direction = up ∧ ¬( ∃ r: Requests •
       r.floor > elevator.position ∧ (r.type =passengerRequest ∨
       r.type = upRequest ) ) ) )

 elevator'.position = elevator.position - 1
 elevator'.direction = down
 elevator'.state = moving
 elevator'.door = closed
 requests' = requests
------------------------------------------------------------------------------------------
```

4.3.2.2 Discussion of the Specification in Z

The elevator control system is a concurrent real time embedded system. A concurrent model is needed to specify this system. The specification in Z starts in a state that satisfies the initial conditions of the system. At each step in the execution of the system, an operation is selected and executed non-deterministically according to the following rule: any operation that is continuously enabled will eventually occur. In this model, two or more operations can be thought of as being concurrent if they are enabled at the same time. This model is specified by the operation *MakeRequests* which adds the new requests to the set of requests currently pending in the elevator system. At any moment in time, a number of requests may be made for the elevator.

The specification in Z is precise and concise. The drawback is that it is relatively hard to understand.

4.3.3 UML Approach

4.3.3.1 Specification in UML

In this section, textual English and UML notations are used to specify and analyze the elevator concurrent control problem. Identifying use cases and behaviors of the system follow the previously overviewed process in Section 2.2.5.

In the requirement of the elevator system, the actor is the passenger. There are two cases to activate this system: either on the floor or in the elevator. Both can happen at the same time. The constraint is that the elevator will service all requests in the same direction. It only changes direction if all requests in the current direction have been serviced completely.

4.3.3.1.1 Identifying Use Cases

Figure 4.27 is the main use case diagram of the elevator control problem. A passenger refers to any person who uses the elevator either on the floor or in the elevator to take a ride. A potential passenger refers to any person who makes a floor request when the elevator is servicing.

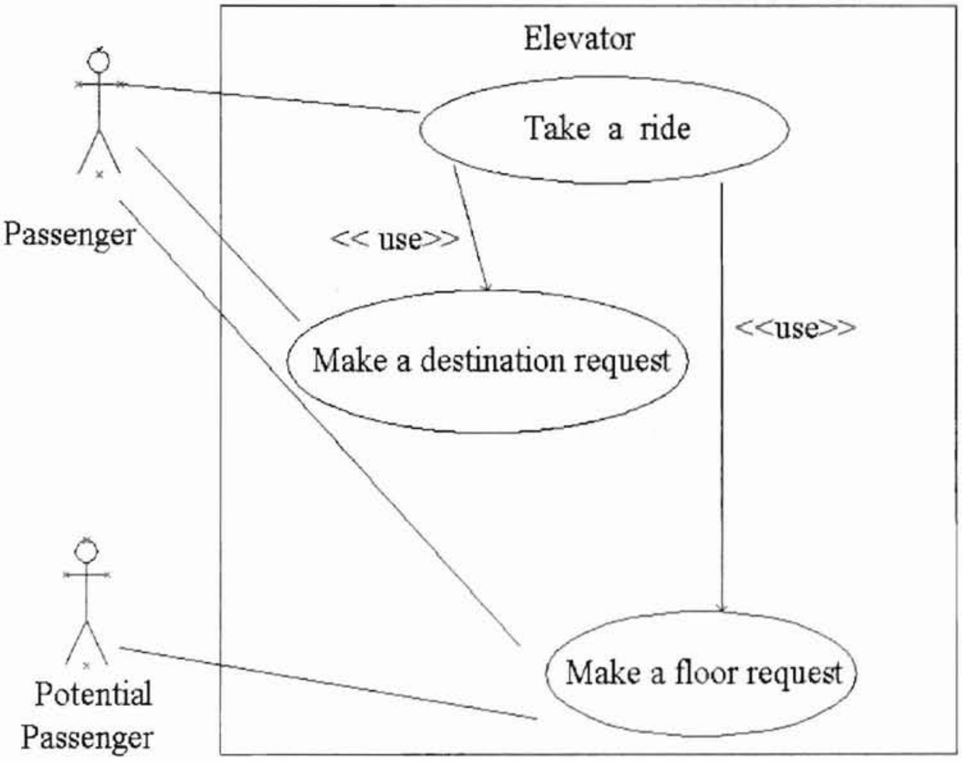


Figure 4.27 Main Use Case Diagram of the Elevator Problem

The following is the documentation of the above three use cases: Make a destination request, Make a floor request, and Take a ride in the elevator system.

1) Make a Floor Request

This use case starts when a passenger pushes the floor button. The elevator controller accepts the request. The elevator controller moves the elevator to the current floor if the elevator is not on this floor. The elevator controller opens the door. The passenger enters the elevator. The elevator door closes. This use case ends.

Alternative Courses: the elevator is idle on the current floor now, the elevator is idle on another floor now, the elevator is servicing the same direction and will pass through this floor, the elevator is servicing the same direction and will not pass through this floor, or the elevator is servicing the other direction.

The sequence diagram scenario of this use case is represented in Figure 4.28.

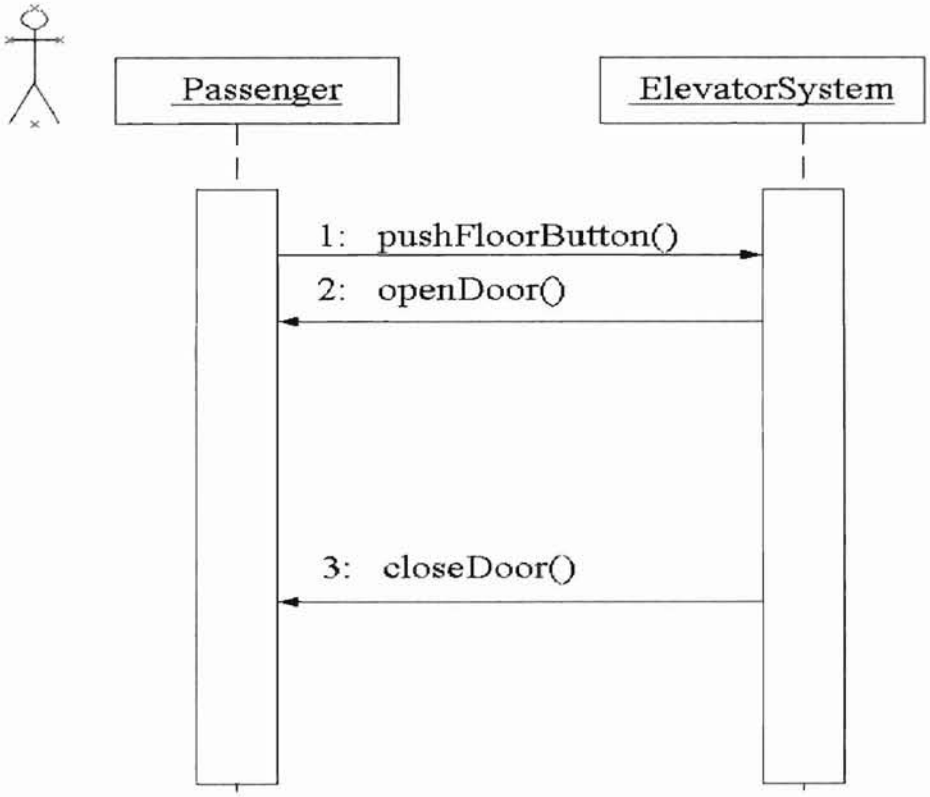


Figure 4.28 Sequence Diagram Scenario of Make a Floor Request

2) Make a Destination Request

This use case starts when a passenger pushes the destination button in the elevator. The elevator controller accepts the request then it moves the elevator to the destination floor and opens the door. The passenger goes out of the elevator. The controller closes the door. This use case ends.

Alternative Courses: the elevator is servicing the other direction, this request will be pending until all requests in the current direction have been serviced.

The sequence diagram scenario of this use case is represented in Figure 4.29.

3) Take a Ride

This use case starts when a passenger makes a floor request and then makes a destination request. This use case ends.

Alternative Courses: a passenger just Make a Floor Request, the elevator just services the floor request.

The sequence diagram scenario of this use case is represented in Figure 4.30

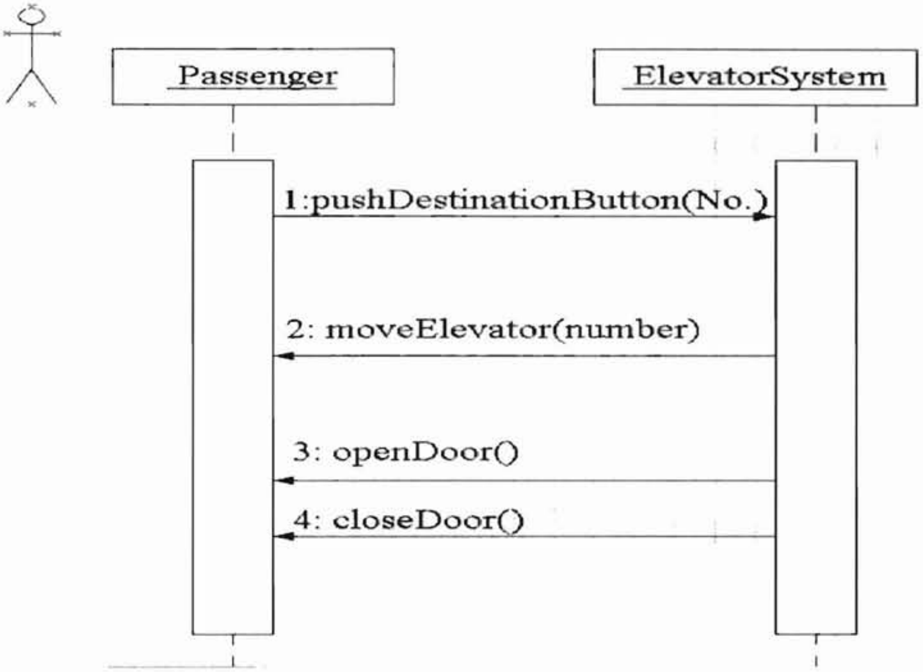


Figure 4.29 Sequence Diagram Scenario of Make a Destination Request

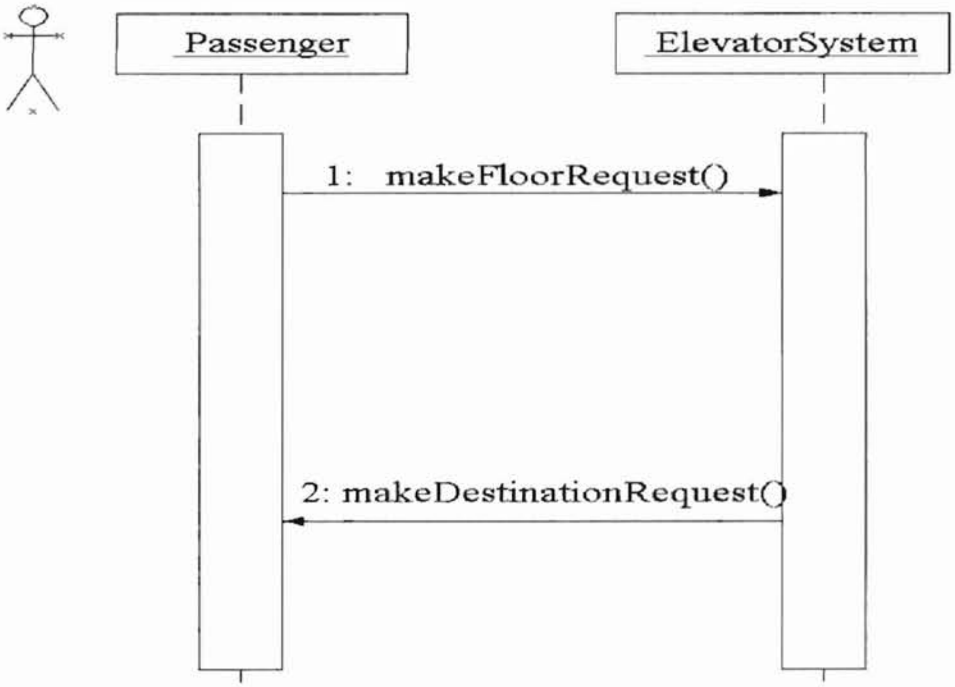


Figure 4.30 Sequence Diagram Scenario of Take a Ride

4.3.3.1.2 Identifying Behaviors of the System

In this section, a collaboration diagram and a state diagram for each use case are drawn to show the behavior of the system. It follows the same conventions as in Section 4.2.3.1.2.

Make a Floor Request

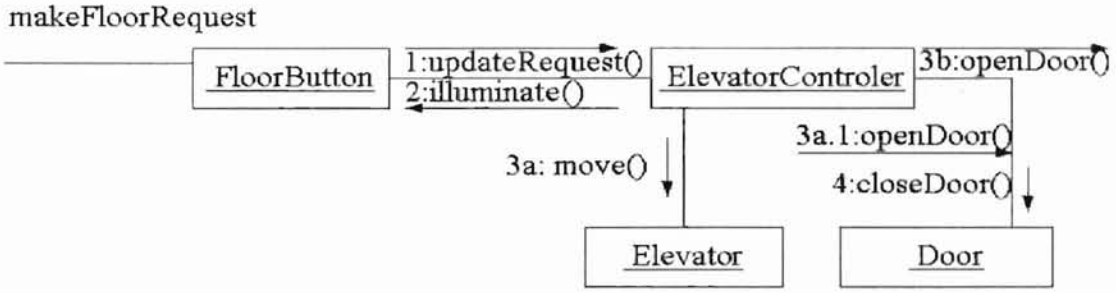


Figure 4.31 Collaboration Diagram of Make a Floor Request

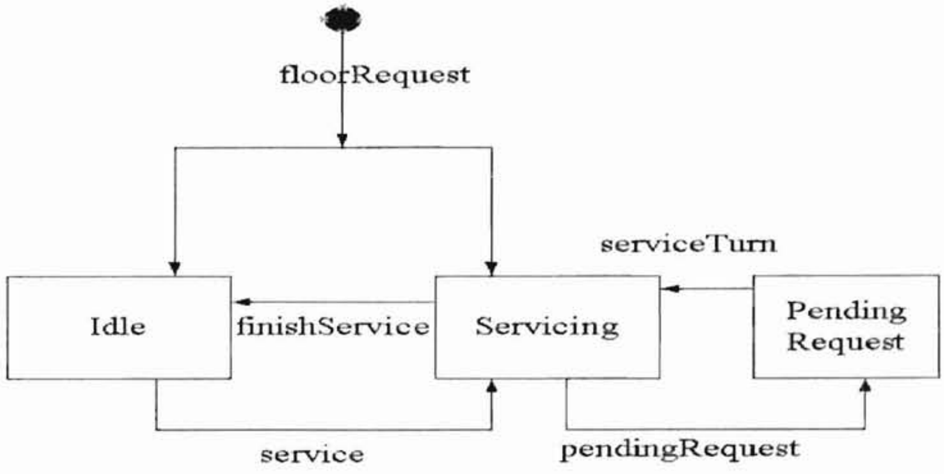


Figure 4.32 State Diagram of Make a Floor Request

Make a Destination Request

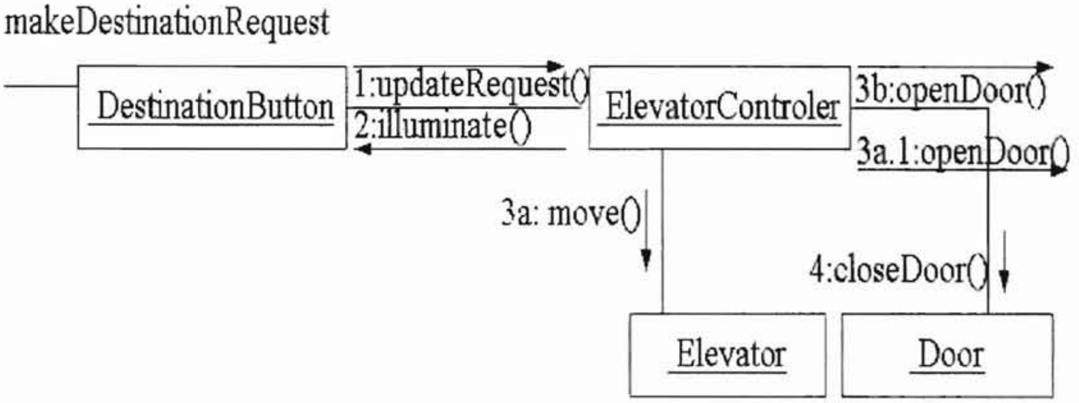


Figure 4.33 Collaboration Diagram of Make a Destination Request

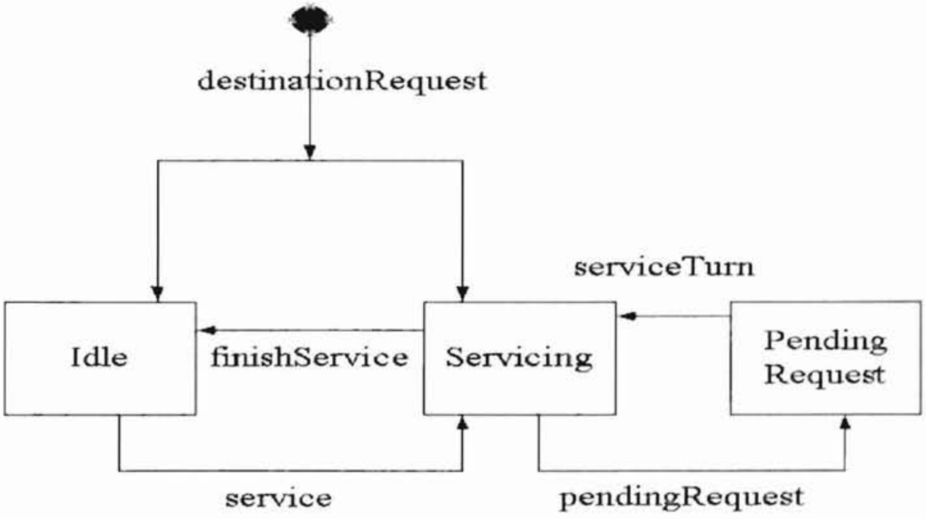


Figure 4.34 State Diagram of Make a Destination Request

Take a Ride

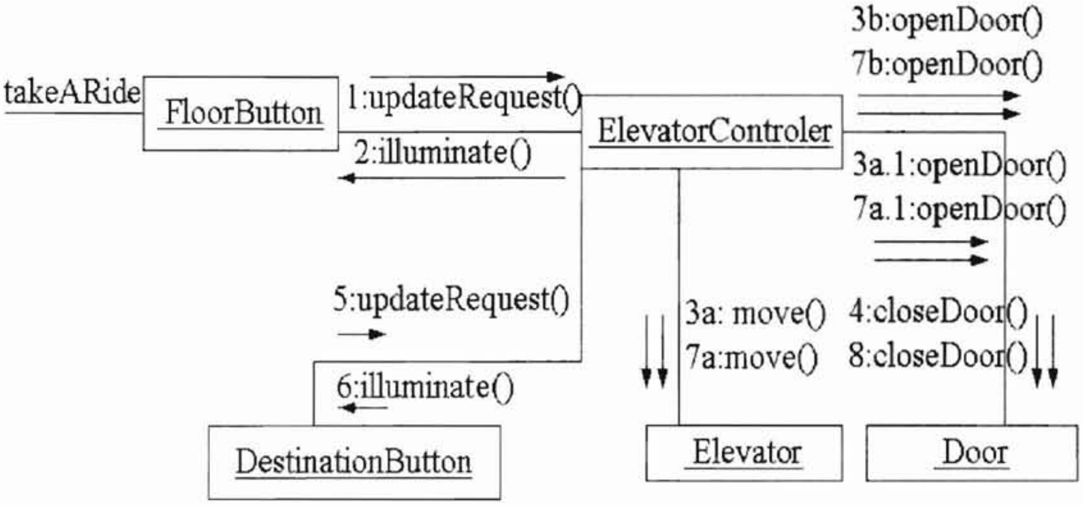


Figure 4.35 Collaboration Diagram of Take a Ride

Figure 4.36 State Diagram of Take a Ride

4.3.3.2 Discussion of the Specification in UML

There are notations, and mechanisms, collaboration diagrams, and state diagrams to describe concurrency in UML. The specification in UML more clearly describes concurrency than the previous specification in Z did.

4.3.4 Comparison of Z and UML Approaches

In this section, the comparison follows the same convention as before (see Section 3.2), i.e., the related facts in the specifications in Z and UML are compared and listed. There is no simple quantitative result to show. Discipline and ambiguity criteria are not compared because there is no additional information provided.

4.3.4.1 Readability

The specification in Z uses schemas as a concurrent model to describe the concurrent elevator control system. It is not very clear and easy to follow or understand. Readers need to carefully analyze the specification. In contrast, the specification in UML is clear using stereotype concurrent syntax and state diagrams.

4.3.4.2 Applicability

The specification in Z does not describe the elevator system in a natural and straightforward manner. Abstract entities and schemas are used to describe the elevator system.

The specification in UML describes the elevator system in a natural and straightforward manner from the user's point of view. Classes are real world objects.

Concurrent events are described using state diagrams. The state diagrams like flowcharts are easy to understand. UML is more applicable to the elevator system than Z.

4.3.4.3 Levels of Abstraction / Expressibility

The specification in Z is typically abstract and it is therefore relatively harder to understand. Entities and schemas are not closely and expressively described. The users of the system are not described. The specification in UML closely and expressively describes the system. It is more natural and easier to understand. The external system is also described.

4.3.4.4 Verifiability

The specification in Z can be proved to be correct using preconditions and post-conditions, but the specification in UML cannot be proved to be correct.

4.3.4.5 Redundancy

The specification in Z is precise without redundancy. In contrast, the specification in UML represents a piece of knowledge more than once from different views.

4.3.4.6 Reusability

Both specifications in Z and UML reuse parts of the respective models to represent other models as the previous case study involving a library system.

CHAPTER V

SUMMARY AND FUTURE WORK

5.1 Summary

In this thesis, the formal specification language Z and the informal object-oriented specification language UML were compared based on a set of criteria from language aspects. Then two classic case studies (the library database system and the elevator concurrent control system) were specified in Z and UML, and were compared based on another set of criteria from application aspects. The results of comparison have been listed in each part of the thesis report. There is no clear "winner" between Z and UML, in the sense that neither Z nor UML dominates in all criteria.

This thesis revealed that UML provides a good approach to modeling the main concepts of the considered cases and the relationships among them using UML use cases. However, it seems that UML is not suitable to the specification phase when we want to precisely address the problem. In contrast, Z seems to be more suitable to the specification phase than the design phase.

Z and UML appear to be complementary. A recommended approach to specifying a problem would be choosing the language after a careful analysis of the problem to be solved, or combining Z and UML in different phases and aspects based on the differences between Z and UML.

5.2 Future Work

Formal methods and object-oriented methods appear to be the main streams of software development. Z and UML are two of the successful methods. Developing and combining Z and UML seem to bc the preferred trends. Quantitative criteria and more complex case studies are needed to compare Z and UML in all aspects thoroughly.

This thesis showed that Z and UML appear to be complementary. Combing Z and UML in different phases and aspects for particular projects is a recommended approach. Methods and rules for combing and or integrating Z and UML are needed.

REFERENCES

[Ardis et al. 96] M. A. Ardis, J. A. Chaves, L. J. Jagadeesan, P. Mataga, C. Puchol, M. G. Staskauskas, and J. V. Olnhausen, "A Framework for Evaluating Specification Methods for Reactive Systems - Experience Report", *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, pp. 378-389, June 1996.

[Bicarregui and Ritchie 95] J. Bicarregui and B. Ritchie, "Invariants, Frames and Postconditions: A Comparison of the VDM and B Notations", *IEEE Transactions on Software Engineering*, Vol. 21, No. 2, pp. 79-89, February 1995.

[Booch et al. 99] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley Longman, Inc., Reading, MA, 1999.

[Bourdeau and Cheng 95] R. H. Bourdeau and B. H. C. Cheng, "A Formal Semantics for Object Model Diagrams", *IEEE Transactions on Software Engineering*, Vol. 21, No. 10, pp. 799–821, October 1995.

[Bowen 96] J. Bowen, *Formal Specification & Documentation Using Z*, International Thomson Computer Press, Boston, MA, 1996.

[Cooke et al. 96] D. Cooke, A. Gates, E. Demirörs, O. Demirörs, M. M. Tanik, and B. Krämer, "Languages for the Specification of Software", *The Journal of Systems and Software*, Vol. 32, No. 1, pp. 269–308, January 1996.

[Cooke and Daniel 94] S. Cooke and J. Daniel, "Let's Get Formal", *Journal of Object-Oriented Programming*, Vol. 7, No. 4, pp. 22 – 24, July–August 1994.

[Diller 90] A. Diller, *Z: An Introduction to Formal Methods*, John Wiley & Sons, West Sussex, England, 1990.

[Evans 94] A. Evans, "Specifying and Verifying Concurrent Systems Using Z", *Proceedings of the Second International Symposium of Formal Methods, Europe,* pp. 366-388, Barcelona, Spain, October 1994.

[France et al. 97] R. B. France, J. M. Bruel, and M. M. Larrondo-Petrie, "An Integrated Object-Oriented and Formal Modeling Environment", *Journal of Object-Oriented Programming*, Vol. 10, No. 7, pp. 16–23, November-December 1997.

[France et al. 98] R. France, A. Evans, K. Lano, and B. Rumpe, "The UML as a Formal Modeling Notation", *Computer Standards & Interfaces*, Vol. 18, No. 5, pp. 325–334, March 1998.

[Harmelen et al. 96] F. van Harmelen, M. Aben, F. Ruiz, J. van de Plassehe, "Evaluating a Formal KBS Specification Language", *IEEE Expert* (see also *IEEE Intelligent Systems*), Vol. 11, No. 1, pp. 56-61, February 1996.

[Hayes 87] I. Hayes, *Specification Case Studies*, Prentice-Hall International, Inc., Upper Saddle River, NJ, 1987.

[Henderson-Sellers and Edwards 94] B. Henderson-Sellers and J. M. Edwards, *Object-Oriented Knowledge: The Working Object.* Prentice Hall, Sydney, Australia, 1994.

[Henderson-Sellers and Firesmith 99] B. Henderson-Sellers and D. G. Firesmith, "Comparing OPEN and UML: The Two Third-Generation OO Development Approaches", *Information and Software Technology*, Vol. 41, No. 3, pp. 139-156, February 1999.

[Kim and Carrington 99] S. K. Kim and D. Carrington, "Visualization of Formal Specifications", *Proceedings of the Sixth Asia Pacific Software Engineering Conference*, pp. 102–109, Takamatsu, Japan, December 1999.

[King and Sorensen 89] S. King and I. H. Sorensen, "From Specification, Through Design to Code: A Case Study in Refinement", In *Formal Methods: Theory and Practice,* edited by P. N. Scharbach, CRC Press, Inc., Boca Raton, FL 1989.

[Lano 90] K. C. Lano, "Z++, An Object-Oriented Extension to Z", *Proceedings of the Fifth Annual Z User Meeting*, pp. 34-46, Oxford, England, December 1990.

[Neil et al. 98] M. Neil, G. Ostrolenk, M. Tobin, and M. Southworth, "Lessons from Using Z to Specify a Software Tool", *IEEE Transactions on Software Engineering*, Vol. 24, No. 1, pp. 15-23, January 1998.

[Schach 97] S. R. Schach, *Software Engineering with JAVA*, Time Mirror Higher Education Group, Chicago, IL, 1997.

[Shroff and France 97] M. Shroff and R. B. France, "Towards a Formalization of UML Class in Z", *Proceedings of the Twenty-first Annual International Computer Software and Applications Conference* (COMPSAC'97), pp. 646–651, Washington D. C., August 1997.

[Spivey 89] J. M. Spivey, *The Z Notation: A Reference Manual*, Prentice Hall, Englewood Cliffs, NJ, 1989.

[Wing 88] J. M. Wing, "A Study of 12 Specifications of the Library Problem", *IEEE Software*, Vol. 5, No. 4, pp. 66–76, July 1988.

[Yourdon and Argila 96] E. Yourdon and C. Argila, *Case Studies in Object-Oriented Analysis and Design*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1996.

APPENDIX A: GLOSSARY

B --- A generic term given to a method of software development, the B-Method, its process and notation and its supporting toolset, the B-Toolkit.

Bag --- Bags sometime are known as families or multi-sets. A bag of things is similar to a set where the order of the elements is not significant; but, unlike a set, the number of occurrences of each object in the bag is significant.

CASE --- Computer-Aided Software Engineering

ESTEREL --- ESTEREL is both a programming language, dedicated to programming reactive systems, and a compiler which translates ESTEREL programs into finite-state machines. It is one of a family of synchronous languages. (http://www-sop.inria.fr/meije/esterel/esterel-eng.html, July 29, 2000).

FOOM --- Formal Object-Oriented Modeling

KBS --- Knowledge-Based System

Larch --- A multi-site project methods, languages, and tools for the practical use of formal specifications (http://wind.lcs.mit.edu/spd/larch).

LOTOS --- Language Of Temporal Ordering Specifications, an international formal specification technique for specifying concurrent and distributed systems.

ModeChart --- ModeChart is a synchronous language designed for the specification of real-time systems.

MOSES --- Methodology for Object-Oriented Software Engineering of Systems, a complete life cycle method that has evolved from and refined the work reported in Henderson-Sellers and Edwards [Henderson-Sellers and Edwards 94].

OMG --- Object Management Group, responsible for the development of object-oriented languages and their standardization

OMT --- Object Modeling Technique

OO --- Object-Oriented
OOM --- Object-Oriented Method

OOSE --- Object-Oriented Software Engineering

OPEN --- OO Process, Environment, and Notation, an informal specification method.

SDL --- Specification and Description Language, a modern, high-level programming language. It is object-oriented, formal, and graphical. SDL is intended for the description of complex, event-driven, and real-time communicating systems.

Software Life Cycle --- Software development generally includes eight phases: requirement establishing, specification analysis, planning, design, implementation, integration and system testing, maintenance, and retirement. These phases are collectively called the software life cycle.

Specification Language --- A necessary prerequisite for efficient and effective communication among users, requirements engineers, and designers/implementers. It also provides frames within which problems can be defined and solved.

UML --- Unified Modeling Language, an informal specification language that was initially created by the merger of ideas mainly from three approaches: Booch, Jacobson's OOSE (Object-Oriented Software Engineering), and Rumbaugh's OMT (Object Modeling Technique).

VDM --- Vienna Development Method, a formal specification language. VDM originated in the research on formal semantics of programming languages at IBM's Vienna Laboratory in the 1960s and 70s, including the VDL and Meta-IV notations. VDM is their modern descendent, now used well beyond the bounds of language semantics in industrial system development as well as academic research. A number of other specification languages extend or were inspired by VDM including VDM-SL, RSL, VVSL and VDM++ (http://www.cs.tcd.ie/FME/FAQ /vdm/part2.html, June 5, 2000).

VFSM --- Virtual Finite State Machine Methodology, a formal specification method.

Z --- A formal specification language that is properly pronounced "zed". The name Z was given by its inventor Jean-Raymond Abrial in honor of the great set theorist Ernst Friedrich Ferdinand Zermelo. It was developed at the Oxford University Computing Laboratory in 1970's [Schach 97].

APPENDIX B: Z NOTATION

This appendix summarizes the standard Z notation as defined by Spivey [Spivey 89]. They appear in the order of set theoretic symbols, logical connectives, schemas, predicates, relations, functions, and sequences.

Set Theoretic Symbols

$\{\ \}, \varnothing$	empty set
$\{\ \ldots\ \}$	set braces
$\#$	cardinality
$\{\ \ldots \mid \ldots\}$	set definition, predicate definition
$\in$	set membership
$\notin$	non-membership
$\cup$	set union
$\cap$	set intersection
$\boldsymbol{P}$	power set
$\boldsymbol{F}$	finite set
$\setminus$	set difference
$\subset$	proper subset
$\subseteq$	subset

Logical Connectives

$\wedge$	and
$\vee$	or
$\neg$	not
$=$	equality
$\neq$	inequality
$\Rightarrow$	implies
$\Leftrightarrow, \equiv$	equivalence
$\bullet$	such that

Schemas

Δ	changeable
Ξ	unchangeable
$\vee$	schema disjunction
$\wedge$	schema conjunction

≅	schema definition

Predicates

∃	existential quantification
∃ !	unique quantification
∀	universal quantification

Relations

↔	relation
dom	domain
ran	range
R^{-1}	reverse relation
<	domain restriction
⩤	domain anti-restriction
>	range restriction
⩥	range anti-restriction
;	relation composition

Functions

→	total function
⤀	partial surjection
α	maplets
⊕	function overriding

Sequences

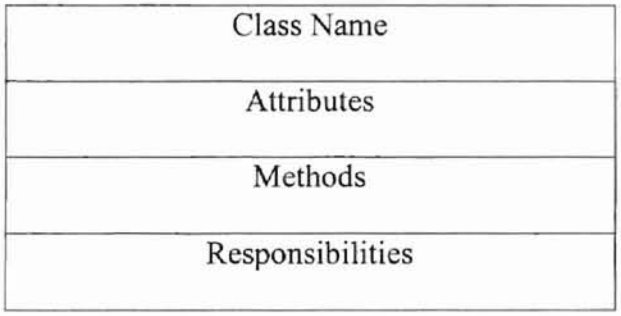
< … >	sequence
front	front
head	head
last	last
rev	reverse
seq	definition
tail	tail

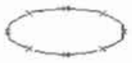
APPENDIX C: UML NOTATION

This appendix summarizes the elements of UML notation: things, relationships, extensibility, and diagrams.

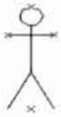
Things: Things include structural things, behavioral things, grouping things, and annotational things.

Structural things include classes, interfaces, collaborations, use cases, active classes, components, and nodes.

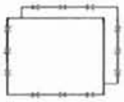




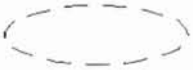
use case



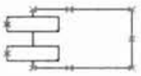
actor



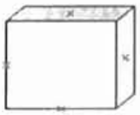
multi-object



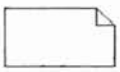
collaboration



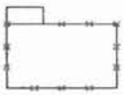
component



Node



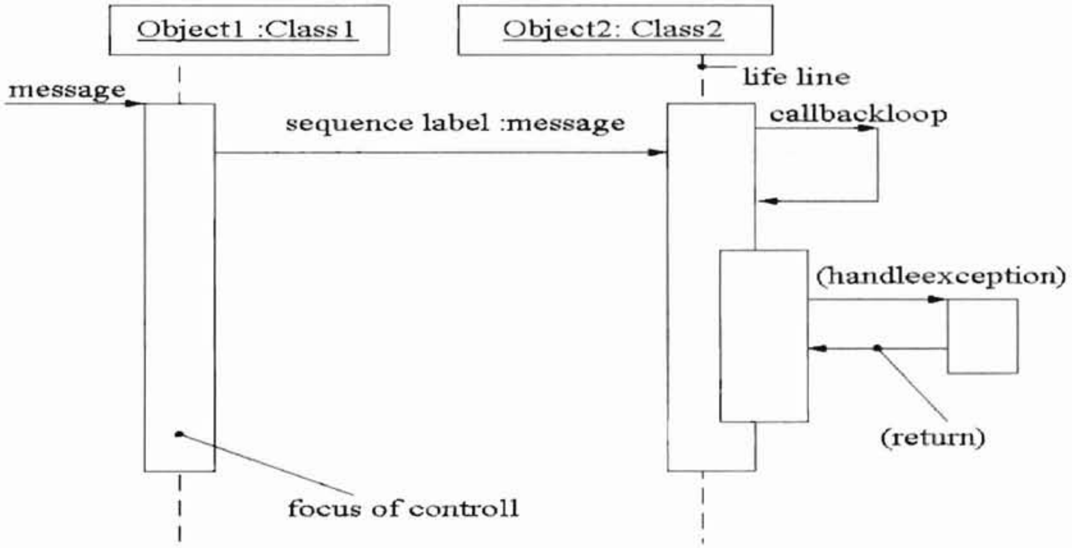
Note



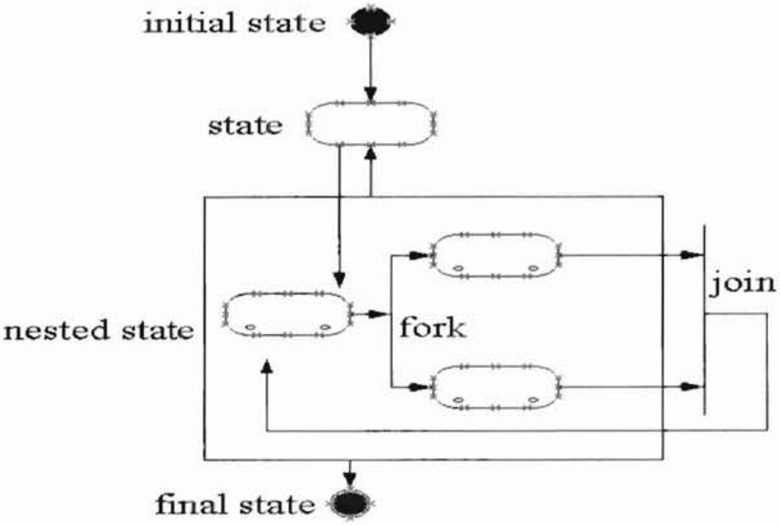
package

Behavioral things include interaction diagrams and state diagrams.

Interaction Diagram:



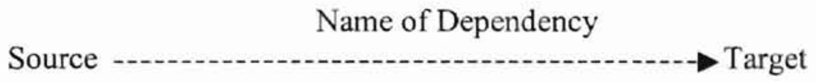
State Diagram:



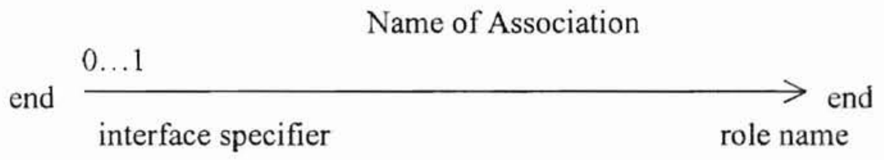
Grouping things include packages. Annotational things include notes. Both are provided in above figures.

Relationships: Relationships include dependency, association, and generation.

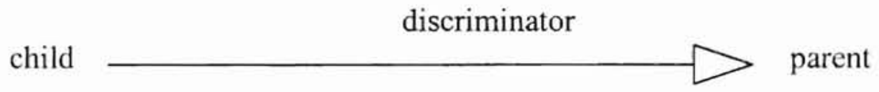
Dependency:



Association:



Generalization:



Extensibility: Extensibility includes stereotypes and tagged values and constraints.

Stereotypes: << … >>

Tagged values and constraints: { … }

Diagrams: Diagrams include class diagrams, object diagrams, use case diagrams, sequence diagrams, collaboration diagrams, statechart diagrams, and activity diagrams.

APPENDIX D: Z TOOL SUPPORT

This appendix summaries some of the support tools available for Z. The detailed descriptions of these tools can be found at: http://www.comlab.ox.ac.uk/archive /z.html.

- Mike Spivey's original style file for writing Z documents using the LaTeX document preparation system (see also the MiKTeX system for Windows users) which is available as zed.sty together with a guide in LaTeX source or compressed PostScript format.
- FuZZ is a Z type-checker which is available commercially, together with an associated fuzz.sty LaTeX style file which has better fonts for the more esoteric Z symbols. The type-checker runs on Sun workstation, PC, and VAX/VMS equipment. It is compatible with zed.sty and the ZTC type-checker LaTeX-Zed input format. It requires the latexsyms.sty style file.
- Object-Z LaTeX Macros for LaTeX 2.09 (oz.sty) and 2e (oz2e.sty) including documentation. zed-cm.sty have been further developed to work with LaTeX2e, together with some documentation files.
- Alcoa: Alloy Constraint Analyzer based on a Z-like language for object models.
- CADiZ provides support for Z using troff and LaTeX on Unix systems and Microsoft Word on PCs from York Software Engineering Ltd.
- Cogito is a methodology and tool set for formal development of software. It uses the Sum specification and development language based on Z notation.

- HOL mechanical theorem proving system is based on Higher-Order Logic and provides some support for Z.
- HIPPO Z type-checker uses ML in compressed Unix tar format. HOL-Z tool is a shallow embedding of Z in the higher-order logic theorem prover.
- JAPE (Just Another Proof Editor) by Bernard Sufrin and Richard Bornat supports typed set theory (Z).
- Mathias Mathematics in Animation Suite includes advice on animating Z and references to papers on Suzan (the Surrey Z Animation project).
- Moby/OZ is a graphical editor to build Z and Object-Z specifications.
- Nitpick is a freely available tool for fully automatically analyzing software specifications in a subset of Z.
- OOZE is an object-oriented Z environment.
- PiZA is a Prolog Z animator.
- Possum is an interpreter/animator of Z.
- PRECC Z with the PRECC compiler tool is available under Unix and MS-DOS.
- ProofPower is an industrial strength Z theorem prover based on higher order logic, which is available from ICL.
- RoZeLink is the Rose Formaliser Link, from Headway Software. It combines the Unified Modeling Language (UML) and Z notations for formal modeling.
- TrueType Z fonts are available under MS Windows (in pkzip format). Zed font is for Windows and Macintosh developed by Richard Jones.
- Wizard is a type-checker for Object-Z specifications in LaTeX.

- ZETA is an open environment based on Z, and provides an integration framework for tools to edit, browse, analyze, and animate Z specifications. Tools currently integrated into ZETA are the type checker ESZ, the Z compiler ZAP, and adaptors for third-party tools such as the prover Isabelle/HOL-Z and the model-checker SMV.
- ZAST (Z Abstract Syntax Tree Viewer) is based on a PreCC Z grammar, and runs under Microsoft Windows 95 and Window NT.
- Z Browser is available under Windows on PCs.
- Zeus WYSIWYG toolset is based on Framemaker and Z/EVES.
- Zola is a commercial WYSIWYG editor, type-checker, and tactical prover for Z from IST, UK.
- ZTC is a Z type-checker. It is available free of charge for educational and non-profit uses. It is intended to be ZRM2 compliant. It accepts LaTeX with zed or oz styles and ZSL - an ASCII version of Z. ZANS is a Z animator. It is a research prototype that is still very crude. Both ZTC and ZANS run on Linux, SunOS 4.x, Solaris 2.x, Windows 95 and NT 4.0. They are available on-line in the directories of ZANS and ZTC.

APPENDIX E: UML TOOL SUPPORT

This appendix summaries some of the support tools available for UML.

- Microsoft Visio® 2000 is a visual language for business. It enables users to communicate effectively with easy-to-assemble drawings and diagrams to create organizational charts and flowcharts, to draw technical schematics and annotate CAD drawings, and to build Web site maps. It can also be used to draw UML. (http://www.microsoft.com/office/visio/)
- MagicDraw is a visual UML modeling and CASE tool designed for the business analyst, software analyst, programmer, QA engineer, documentation writer, or corporate executive. This tool allows the developer to draw, design, and view UML diagrams of object oriented systems. The development tool is oriented towards one user desktop development, and it integrates with Rational Rose 98 easily. MagicDraw runs on a wide variety of operating systems such as Windows 98/NT, Solaris, OS/2, Linux, HP-UX, and AI. (http://www.nomagic.com/magicdrawuml/)
- Visual Thought enables users to create technical and documentation graphics, network diagrams, flowcharts, software design diagrams (UML, Booch, OMT, Objectory, etc.), circuit & logic diagrams, and business and presentation graphics. (http://www.confluent.com/)
- The ArgoUML is an object oriented design tool for designers when they are making design decisions. This tool follows UML standards. (http://argouml.tigris.org/)

- GDPro is a complete UML visual modeling solution for accelerating software design and collaborative development. It allows application developers to create and communicate across the enterprise using simple, comprehensive visual blueprints of Java, C++, and IDL. (http://www.advancedsw.com/)
- I-Logix is a provider of design automation solutions for developers of real-time embedded systems. I-Logix' products allow engineers to graphically model the behavior and functionality of complex embedded systems, analyze and validate system behavior through simulation and animation, and facilitate design level debugging. (http://www.ilogix.com/fs_home.htm)
- The Objecteering CASE tool concentrates on the real force within business. It is the only European player to contribute to the evolution of UML. (http://www.softeam.fr/us/index.htm)
- ObjeXion UML Prototyping Suite for Rational Rose™ is for prototyping UML models, and it is tightly integrated with Rational Rose. (http://www.objexion.com/)
- PLATINUM Paradigm Plus is a UML-based component modeling tool that enables organizations to design and visualize complex applications. Paradigm Plus also provides round-trip engineering support without data loss or code markers for the most popular component software development tools including Java, Microsoft Visual Studio 98, PLATINUM Aion, Forte, and PowerBuilder. (http://www.cai.com/products/platinum/appdev/pplus_ps.htm)
- Rational Rose is a leading visual modeling tool in the UML field. The three major contributors constituted the Rational Rose to develop and standardize UML. (http://www.rational.com/products/rose/index.jsp)

- Princeton Softech's Select Enterprise® is an advanced tool set for modeling, designing, and building enterprise applications. (http://www.princetonsoftech.com/)
- Software through Pictures® for the Unified Modeling Language (StP/UML) is the implementation of UML 1.1. StP/UML is a graphical, object-oriented modeling environment to analyze and design critical systems and business applications. (http://www.aonix.com/Products/SMS/StP_UML.html)
- Telelogic Tau UML Suite is a multi-user solution for object-oriented application development, supporting UML 1.3. Its team modeling features enable the management of object-oriented modeling components as corporate assets that drive the analysis, design, and construction of large-scale applications. (http://www.telelogic.com/solution/tools/Uml.asp)

VITA

Bingdong Li

Candidate of the Degree of

Master of Science

Thesis: A COMPARISON OF Z AND UML: TWO CASE STUDIES

Major Field: Computer Science

Biographical:

Personal Data: Born in Henan, P. R. China, on December 17, 1965, son of Mr. Zhixiang Li and Ms. Lanying Niu.

Education: Received Bachelor of Science in Biology from Beijing Agricultural University, Beijing, P. R. China in July 1989; received Master of Science in Biology from the Graduate School of Chinese Academy of Agricultural Science, Beijing, P. R. China in July 1996; completed the requirements for the Master of Science Degree in Computer Science at the Computer Science Department at Oklahoma State University in December 2000.

Experience: Employed by Beijing Agricultural Bureau, August 1996 to April 1998, by Oklahoma State University, Plant and Soil Science Department, September 1998 to July 1999.