CONSISTENCY CHECKING IN MULTIPLE UML STATE DIAGRAMS USING SUPER STATE ANALYSIS

by

MOHAMMAD N. ALANAZI

B.S., King Saud University, Riyadh, Saudi Arabia, 1999
M.S., The American University, Washington, DC, 2003

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2008

Abstract

The Unified Modeling Language (UML) has been designed to be a full standard notation for Object-Oriented Modeling. UML 2.0 consists of thirteen types of diagrams: class, composite structure, component, deployment, object, package, activity, use case, state, sequence, communication, interaction overview, and timing. Each one is dedicated to a different design aspect. This variety of diagrams, which overlap with respect to the information depicted in each, can leave the overall system design specification in an inconsistent state.

This dissertation presents *Super State Analysis* (*SSA*) for analyzing UML multiple state and sequence diagrams to detect the inconsistencies. *SSA* model uses a transition set that captures relationship information that is not specifiable in UML diagrams. The *SSA* model uses the transition set to link transitions of multiple state diagrams together. The analysis generates three different sets automatically. These generated sets are compared to the provided sets to detect the inconsistencies. Because *Super State Analysis* considers multiple UML state diagrams, it discovers inconsistencies that cannot be discovered when considering only a single UML state diagram. *Super State Analysis* identifies five types of inconsistencies: valid super states, invalid super states, valid single step transitions, invalid single step transitions, and invalid sequences.

CONSISTENCY CHECKING IN MULTIPLE UML STATE DIAGRAMS USING SUPER STATE ANALYSIS

by

MOHAMMAD N. ALANAZI

B.S., King Saud University, Riyadh, Saudi Arabia, 1999
M.S., The American University, Washington, DC, 2003

A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2008

Approved by:

Major Professor
Dr. David A. Gustafson

Abstract

The Unified Modeling Language (UML) has been designed to be a full standard notation for Object-Oriented Modeling. UML 2.0 consists of thirteen types of diagrams: class, composite structure, component, deployment, object, package, activity, use case, state, sequence, communication, interaction overview, and timing. Each one is dedicated to a different design aspect. This variety of diagrams, which overlap with respect to the information depicted in each, can leave the overall system design specification in an inconsistent state.

This dissertation presents *Super State Analysis* (*SSA*) for analyzing UML multiple state and sequence diagrams to detect the inconsistencies. *SSA* model uses a transition set that captures relationship information that is not specifiable in UML diagrams. The *SSA* model uses the transition set to link transitions of multiple state diagrams together. The analysis generates three different sets automatically. These generated sets are compared to the provided sets to detect the inconsistencies. Because *Super State Analysis* considers multiple UML state diagrams, it discovers inconsistencies that cannot be discovered when considering only a single UML state diagram. *Super State Analysis* identifies five types of inconsistencies: valid super states, invalid super states, valid single step transitions, invalid single step transitions, and invalid sequences.

Table of Contents

List of Figures ix

List of Tables xi

Acknowledgements xii

Dedication xiii

CHAPTER 1 - Introduction 1

1.1 UML Diagrams 1

1.1 Diagrams Description 2

1.1.1 Class Diagram 2

1.1.2 Object Diagram 3

1.1.3 Component Diagram 3

1.1.4 Composite Structure Diagram 4

1.1.5 Deployment Diagram 4

1.1.6 Package Diagram 4

1.1.7 State Diagram 5

1.1.8 Activity Diagram 5

1.1.1 Use Case Diagram 6

1.1.2 Sequence Diagram 6

1.1.3 Interaction Overview Diagram 7

1.1.4 Communication Diagram 8

1.1.5 Timing Diagram 8

1.2 The Problem 8

1.3 Proposed Solution 10

1.4 The Hypothesis 10

CHAPTER 2 - Literature Review 12

2.1 Introduction 12

2.2 Transformation 13

2.3 Consistency Rules 14

2.4 Formalism 15

CHAPTER 3 - Super State Analysis (*SSA*) Approach..17

3.1 The Super State..17

3.2 Super State Analysis 17

3.3 Comparisons 21

3.4 The Transition Matrix...22

3.5 The Transition Set...23

3.5.1 Transition Set Types 24

3.5.2 Example 24

3.6 Inconsistency Detection...27

3.6.1 State Inconsistencies 27

3.6.2 Single Step Transitions Inconsistencies..27

3.6.3 Sequence Inconsistencies...28

CHAPTER 4 - Case Study I (Library Example)..30

4.1 Description...30

4.2 The Library example invariant 32

4.3 Analysis 33

4.4 Some inconsistencies found in the library example..................................37

CHAPTER 5 - Case Study II (University Example) 39

5.1 Description...39

5.2 The State Diagrams for University Model (UM)......................................44

5.3 Some Invariants for UM 45

5.4 Analysis 46

5.5 Inconsistency Discussion for UM..57

5.5.1 Super State Inconsistencies...57

5.5.2 Single Step Transitions Inconsistencies..58

5.5.3 Sequence Inconsistencies...58

CHAPTER 6 - Scalability...60

6.1 Paired Transitions Technique 61

6.2 Object Reduction Technique 63

6.3 State Reduction Technique..64

6.4 Limit the number of steps Technique 65

CHAPTER 7 - Specification and Implementation........66
7.1 Specification........66
7.2 Formalization of *Super State Analysis (SSA)*........66
7.3 Implementation........78
7.3.1 Tool Description........78
7.3.2 Tool Example........79
CHAPTER 8 - Conclusion........84
References........87

List of Figures

Figure 1.1 UML Diagrams.. 2

Figure 1.2 Example of State Diagram.. 5

Figure 1.3 Example of Sequence Diagram 7

Figure 3.1 *SSA* Model 19

Figure 3.2 State Diagram for Class X.. 22

Figure 3.3 State Diagram for Class Y.. 22

Figure 3.4 State Diagram for Customer.. 25

Figure 3.5 State Diagram for Account... 25

Figure 4.1 Class Diagram for the library example... 31

Figure 4.2 State Diagram for Patron.. 31

Figure 4.3 State Diagram for Book.. 32

Figure 5.1 Class Diagram for Univeristy Model 40

Figure 5.2 State Diagram for Enrollment 41

Figure 5.3 State Diagram for Teaching.. 42

Figure 5.4 State Diagram for Student 42

Figure 5.5 State Diagram for Instructor... 43

Figure 5.6 State Diagram for Section 43

Figure 5.7 State Diagram for Room... 44

Figure 5.8 A Sequence Diagram for a Class Enrollment.................................... 59

Figure 6.1 Reduced State Diagram for Student 64

Figure 7.1 The relationship between R and G 74

Figure 7.2 The relationship between T1, H1, and H2 75

Figure 7.3 The relationship between T2, H3, and H4 77

Figure 7.4 *SSA* Tool Architecture 79

Figure 7.5 Transition Set File 80

Figure 7.6 Sequence for returning overdue book 81

Figure 7.7 Tool output for Figure 7.6 sequence diagram 81

Figure 7.8 A corrected sequence for overdue book 82

Figure 7.9 Tool output for Figure 7.8 sequence diagram 82

Figure 7.10 Execution time for 1 Book and 2 Patrons 83

Figure 7.11 Execution time for 2 Books and 1 Patron 83

Figure 7.12 Execution time for 2 Books and 2 Patrons 83

List of Tables

Table 3.1 Description of each component involved in *SSA* Model 20

Table 3.2 Super state transition matrix T_1 23

Table 4.1 Portion of A_1 34

Table 4.2 Transition set for Library Example.. 35

Table 4.3 Portion of A_2 36

Table 5.1 Information about the University Model 39

Table 5.2 Transition Set for University Model.. 47

Table 5.3 Transition Set for University Model.. 49

Table 5.4 Portion of A_1 for the transition set in Table 5.2.. 54

Table 5.5 Portion of A_2 for the transition set in Table 5.2.. 55

Table 6.1 Effect of object selection on the total number of super states 62

Table 6.2 Behavior of independent objects in transition set... 63

Acknowledgements

First of all, thanks are purely due to Allah for giving me the strength and confidence to realize and achieve my goals.

I would like to express my deep appreciation for my major professor Dr. David A. Gustafson for his patience, guidance, and encouragement during the journey of this research. Thanks for all the hours of wonderful discussion and for supporting me in so many ways. Thank you for sharing your knowledge and time with me.

I would like to thank my committee members: Dr. William J. Hankley, Dr. Mitchell L. Neilsen, and Dr. Fayez Husseini for their support and for graciously accepting to serve on my committee. I would also like to thank Dr. D. V. Satish Chandra for serving as a chair for my final exam.

My heartfelt gratitude goes to my parents, brothers, and sisters for their inspiration, support, and patience. Their love and constant support have been a great help throughout years.

I would like to thank all the people and loved ones who have helped me during my study in the United States. The most sincere thanks go to my dear wife and my kids Alaa, Bassam, and Jory who made a difficult task easier through their prayers, patience and support.

Finally, I would like to thank Imam University in Saudi Arabia for providing the financial support throughout my graduate studies.

Dedication

To my parents

To my family

CHAPTER 1 - INTRODUCTION

1.1 UML Diagrams

The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML is a graphical language for represent software designs. It provides several diagram types to capture different aspects of design. UML 2.0 specification has thirteen standard diagrams. These diagrams can be divided into two groups: structural diagrams, which model the organization and the structure of a system, and behavioral diagrams, which model the behavior of a system. Figure 1.1 shows the class diagram of the UML diagrams.

Structural Diagrams

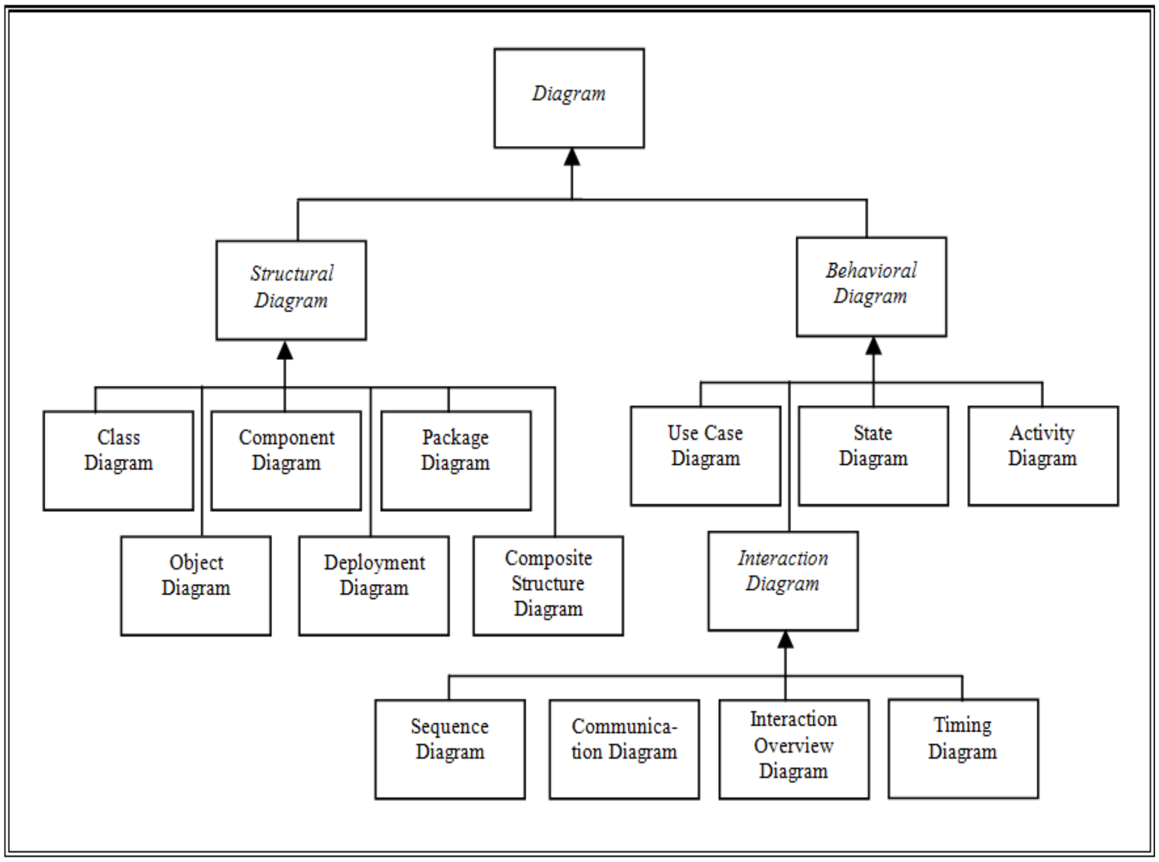
- Class Diagram
- Object Diagram
- Component Diagram
- Deployment Diagram
- Package Diagram
- Composite Structure Diagram

Behavioral Diagrams

- Use Case Diagram
- Sequence Diagram
- State Diagram
- Activity Diagram

- Communication Diagram
- Interaction Overview Diagram
- Timing Diagram

Figure 1.1 UML Diagrams



1.1 Diagrams Description

1.1.1 Class Diagram

A Class diagram represents the static structure of the classes and their relationships (e.g., association, inheritance, aggregation) in a system. The class diagram shows the operations and the attributes of each class. A class is divided into three components: class name, attributes, and operations. The Class diagram is one of the most widely used diagrams from the UML specification. Part of the popularity of class diagrams stems from the fact that many UML case tools can auto-generate code in a variety of languages, including Java, C++, and C#, from these models. These tools can synchronize models and code, reducing the workload, and can also generate class diagrams from object-oriented code.

1.1.2 Object Diagram

An Object diagram shows instances instead of classes. The object diagram describes how the classes interact with each other at runtime in the actual system. The object diagrams are useful for explaining small part of a system with complicated relationships, especially recursive relationships. It shows the relationship between instances of classes at some point in time.

1.1.3 Component Diagram

A component diagram describes how a software system is divided into physical components and shows the dependencies between these components. The component diagram shows the structural relationships between the components of a system. The component diagram also describes the organization of physical software components, including source code, run-time (binary) code, and executables. Physical components include, for example, files, headers, link libraries, modules, executables, or packages. Component diagrams can be used to model and document any the architecture of a system.

1.1.4 Composite Structure Diagram

Composite structure diagram is a structural diagram that shows the internal structure of a class and the collaborations that this structure makes possible. A composite structure is a set of interconnected elements that collaborate at runtime to achieve some purpose. Each element has some defined role in the collaboration. A composite structure diagram is similar to a class diagram, but it describes individual parts instead of whole classes.

1.1.5 Deployment Diagram

The deployment diagram shows the physical configurations of software and hardware. A deployment diagram models the hardware used in implementing a system and the association between those hardware components. Deployment diagrams give a picture of the physical resources in a system, including nodes, components, and connections. The deployment diagram shows the hardware for the system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another.

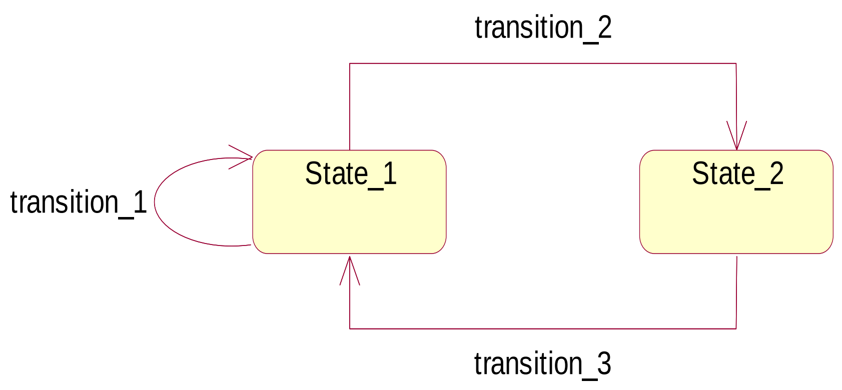
1.1.6 Package Diagram

Packages are UML constructs that allow organizing the model elements into groups to make UML diagrams simpler and easier to understand. A package diagram describes how a system is divided into logical groupings by showing the dependencies among these groupings. The package diagram is most common on use case diagrams and class diagrams because these models have a tendency to grow.

1.1.7 State Diagram

State diagrams, (a.k.a statechart diagrams, state machine diagrams, and state transition diagrams), are used to describe the various states that a class can go through and the events that cause a state transition. Each object has behaviors and state. The state of an object depends on its current activity or condition. A state diagram shows the possible states of the class and the transitions that can make a change in state. State diagrams typically model the transitions within a single class. Figure 1.2 shows an example of a simple state diagram.

Figure 1.2 Example of State Diagram



1.1.8 Activity Diagram

An activity diagram shows the behavior with control structure. An activity represents an operation on some class in the system that results in a change in the state of the system. Activity diagrams and state diagrams are related. The Activity diagram is a variation of the state diagram where the states represent operations, and the transitions represent the activities that happen when the operation is complete. However, an activity diagram focuses on the flow of activities involved in a single process. The activity diagram shows how those activities depend on one another. UML activity diagrams are the object-oriented equivalent of flow charts and data flow diagrams (DFDs) from structured development.

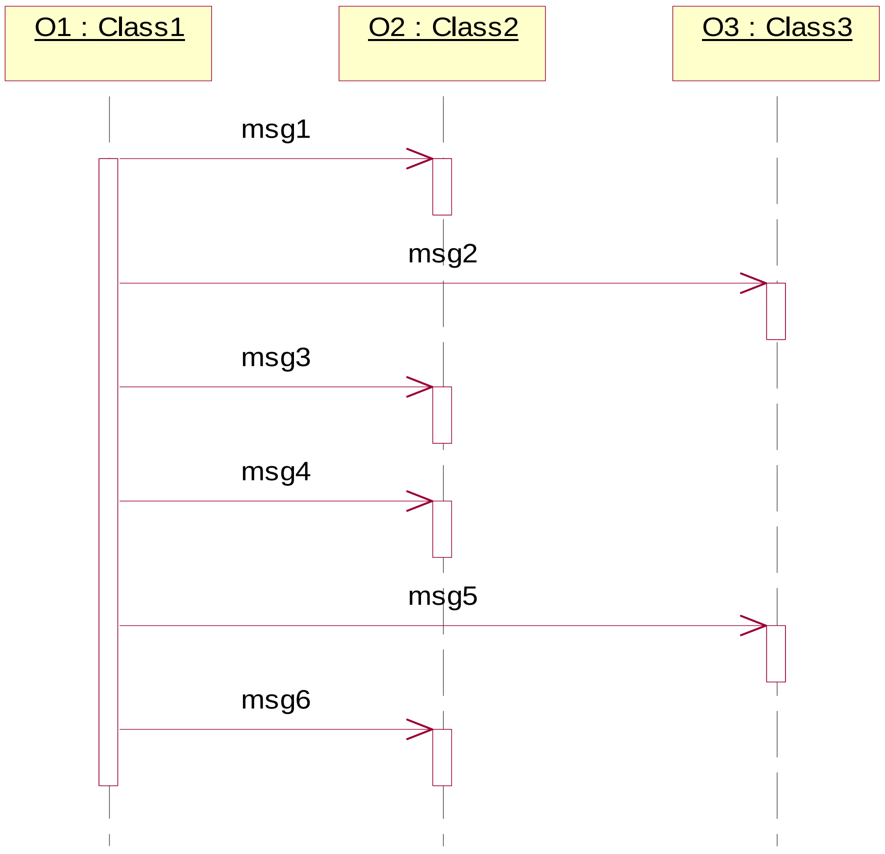
1.1.1 Use Case Diagram

A use case is used to obtain system requirements from a user's perspective. Use case diagrams describe what a system does. The use case diagram emphasizes is on what a system does rather than how. Use Case diagrams identify the functionality provided by the system (use cases), the users who interact with the system (actors), and the relationship between the users and the functionality.

1.1.2 Sequence Diagram

A sequence diagram is an interaction diagram that describes interactions among classes in terms of an exchange of messages over time. Sequence diagrams are organized according to time. The time progresses as you go down the page. The classes involved in the message are listed from left to right according to when they take part in the message sequence. A sequence diagram shows, as parallel vertical lines, different objects that live simultaneously, and, as horizontal arrows, the messages exchanged between them, in the order in which they occur. A sequence diagram describes one possible scenario of the system. Figure 1.3 shows an example of sequence diagram.

Figure 1.3 Example of Sequence Diagram



1.1.3 Interaction Overview Diagram

The interaction overview diagram focuses on the overview of the flow of control of the interactions. An interaction overview diagram is a variant of an activity diagram which overviews the control flow within a system. UML interaction overview diagrams combine elements of activity diagrams with sequence diagrams to show the flow of program execution. The interaction overview diagrams are activity diagrams in which the activities are replaced by little sequence diagrams.

1.1.4 Communication Diagram

A Communication diagram (formally known as collaboration diagrams) describes the interactions between objects or parts in terms of sequenced messages. The collaboration diagram is used to show how objects in a system interact over multiple use cases. The collaboration diagram contains the same information as sequence diagrams, but they focus on object roles instead of the times that messages are sent. Because there is no explicit representation of time in collaboration diagrams, the messages are labeled with numbers to denote the sending order. A communication diagram shows instances of classes, their interrelationships, and the message flow between them. Communication diagrams typically focus on the structural organization of objects that send and receive messages.

1.1.5 Timing Diagram

A timing diagram is used to describe the behaviors of one or more objects throughout a given period of time. Timing diagrams are a specific type of interaction diagram where the focus is on timing constraints. A timing diagram is a special form of a sequence diagram. The differences between a timing diagram and a sequence diagram are that the axes are reversed so that the time is increased from left to right and the lifelines are shown in separate compartments arranged vertically. Timing diagrams are often used to design embedded software.

1.2 The Problem

Unified Modeling Language (UML) has been widely used as a standard language for modeling the software. UML 2.0 [OM06] consists of thirteen types of diagrams: class, composite structure, component, deployment, object, package, activity, use case, state, sequence, communication, interaction overview, and timing. Each diagram is dedicated to a different design aspect. Many different UML diagrams are usually involved in software development. Using more than one diagram to design a system is necessary but can leave the system in an inconsistent state and hence produce errors. Finding inconsistencies in software design before the design is implemented is very important. “Error detection and correction in the design phase can reduce total costs and time to market” [PI03].

A consistency problem may arise due to the fact that some aspects of the model will be described by more than one diagram. Hence, we should pay more attention to the consistency in the early phases of the system development and it is important that the consistency of a system should be checked before implementing it [LI03]. To avoid such errors, we should check the consistency among the diagrams and make sure that the diagrams are consistent.

Many researchers found that the problem of ensuring consistency between UML diagrams has not been solved yet [EG01]. The UML specification does not enforce many consistency requirements between the information contained in the sequence and state diagrams. While this does allow for greater flexibility in how UML can be used, it can lead to inconsistent views of the system being modeled. “The problem of relating state-based intraagent (or intraobject) behavioral descriptions with scenario-based interagent (interobject) descriptions has recently focused much interest among the software engineering community” [BO05]. Identifying inconsistencies between UML diagrams can help the developers to find errors and fix them at early stages. Furthermore, current UML CASE-tools (e.g. Rational® Software Architect [AR08]) provide poor support for maintaining consistency between UML diagrams. So, helping to solve this problem can make a great contribution to the software development process.

1.3 Proposed Solution

The information in UML diagrams are related to each other and represent different views of a system. Hence, they can be validated against each other. Given a state diagram, researchers [LI03] have shown how to validate it against a sequence diagram. On the other hand, given a sequence diagram, it can be validated against a state diagram [DU00, SH06]. In this dissertation, I am proposing a new approach to check the consistency between multiple state diagrams and one or more sequence diagrams using *Super State Analysis* (SSA) to discover the inconsistencies.

Super State Analysis is used to evaluate consistency between multiple state diagrams and the sequence diagrams. *Super State Analysis* helps also to identify the invalid sequence diagrams. The analysis discovers inconsistencies that cannot be detected when considering only a single state diagram. This analysis gives a great contribution to solving the consistency problem between multiple state diagrams and sequence diagrams.

1.4 The Hypothesis

The *Super State Analysis (SSA)* handles the inconsistencies in UML multiple state diagrams and sequence diagrams. *Super State Analysis* may identify inconsistencies in states (see 1 and 2 below), single step transitions (see 3 and 4 below), and sequences (see 5 below). Because *Super State Analysis* considers multiple UML state diagrams, it discovers some inconsistencies that cannot be discovered when considering only a single UML state diagram. *Super State Analysis* does not handle other inconsistencies that deal with other UML diagrams other than state and sequence diagrams. The scope of this dissertation is only UML state and sequence diagrams.

Specifically, *Super State Analysis* may identify the following five types of inconsistencies that are related to state and sequence diagrams:

Inconsistency in states

1. Valid super states
2. Invalid super states

Inconsistency in single step transitions

3. Valid single step transitions
4. Invalid single step transitions

Inconsistency in sequences

5. Invalid sequences

CHAPTER 2 - LITERATURE REVIEW

2.1 Introduction

There are several different approaches that have been proposed to perform consistency checking between UML diagrams. Some approaches use transformation to convert one diagram to another [EG01, WA05, ST04, WA03, SH06, PI03] while others detect the inconsistencies by comparing one diagram to another using consistency rules [LI03, EG06]. Moreover, many approaches use formalism, such as OCL and Z, to enforce the consistency [DU00, GO03, KR00, KI04].

Almost all approaches focus on all or some of six types of UML diagrams. Namely use case class, object, sequence, collaboration, and statechart diagram. Ludwik Kuzniarz et al. [KU03] studies the consistency between use case, class, sequence, and statechart diagram. Alexander Egyed [EG01] studies the consistency between class, object, sequence, collaboration, and statechart diagram. Hassan Gomaa et al. [GO03] studies use case, class, sequence, and statechart diagram. Ragnhild Van Der Straeten et al. [ST04] studies the consistency between three diagrams: class, sequence, and statechart diagram. [LI03, DU00, WA05, SH06] study the consistencies between sequence and statechart diagram. Zs. Pap et al. [PA01] studies the class diagram and statechart diagram.

The researchers pay the attention to enforce consistency between only two diagrams (e.g. single sequence diagram vs. single statechart diagram). However, my approach is unique in that I am proposing a new approach to check the consistency between multiple state diagrams and one or more sequence diagrams using a transition matrix. Moreover, the approach focuses on multiple state diagrams instead of a single state diagram.

2.2 Transformation

The consistency checking in the transformational approach is done in two steps. First, the UML diagrams are converted to interpreted diagrams. Second, the interpreted diagrams are compared to each other to detect the inconsistencies.

Alexander Egyed [EG01] presents a transformation-based approach to consistency checking. He defines a set of model transformation rules to enable the conversion of one UML diagram into another. He also defines a set of comparison rules to compare the transformed diagram with an existing one of the same type. For example, to check for inconsistencies between a sequence diagram and a class diagram, they first transform the sequence diagram into an interpreted class diagram. The interpreted class diagram is then compared with the existing class diagram. This approach needs two sets of rules: transformation rules and consistency rules. If one diagram cannot transform to another, then both diagrams transformed to an intermediate diagram to make the comparison.

Hongyuan Wang et al. [WA05] propose an approach that checks the consistency between sequence diagrams and state diagrams. The approach converts statecharts using Finite State Processes and transforms sequence diagram to messages trace. They use an existing tool LTSA to support their approach. However, the approach considers only single sequence diagram and single stateschart diagram.

Wuwei Shen et al. [SH06] propose to build a message graph from a statechart diagram and then go through the graph based on the sequence of the messages retrieved from a sequence diagram to find any inconsistency between these two diagrams. Based on this method, a tool called ICER is developed to provide software developers with automatic consistency checking in the dynamic aspects of a model. However, the approach considers only single statechart vs. single sequence diagram.

Orest Pilskalns et al. [PI03] present an approach that combines structural and behavioral UML representations in order to derive and execute test cases to validate a UML model. They develop a method for encapsulating the behavioral aspects (i.e. message paths between objects) that exists in sequence diagrams into a directed acyclic graph. The objects in the graph are then associated with class attribute/parameter values which are used to generate and execute test cases. Their approach would require OCL object constraints to be written.

2.3 Consistency Rules

In this approach, the consistency is checked using set of consistency rules. The diagrams are compared to each other directly without transformation or formalism.

Boris Litvak et al. [LI03] present an approach to check the consistency between UML sequence and state diagrams. They created the BVUML (Behavioral Validator of UML) tool which automates the behavioral validation process. Their approach associates states with only one object lifeline in the sequence diagram so a single run of the tool validates consistency for only one object. Therefore the tool must be run multiple times in order to check the consistency of an entire sequence diagram.

Alexander Egyed [EG06] introduce an approach for quickly, correctly, and automatically deciding what consistency rules to evaluate when a model changes. The approach does not require consistency rules with special annotations. Instead, it treats consistency rules as black-box entities and observes their behavior during their evaluation to identify what model elements they access. The UML/Analyzer tool integrated with Rational Rose are fully implements this approach. It was used to check 24 types of consistency rules. The author found that the approach provided design feedback correctly and required, in average, less than 9 ms evaluation time per model change with a worst case of less than 2 seconds at the expense of a linearly increasing memory need. However, my approach compares multi statechart diagrams with sequence diagrams.

2.4 Formalism

Since UML is not precise enough, some researchers formalize the UML diagrams to some formal languages (e.g. Z). They then compare this formalism to detect the inconsistencies between the diagrams.

Yves Dumond et al. [DU00] show that it is possible to integrate semi-formal and formal methods for the dynamic behavior of the UML models. The objective is to favor the integration of formal techniques in the actual practice of software engineering. They introduce an approach to formalize sequence diagrams and verify coherence with the statechart diagrams. The approach translates the UML sequence diagrams into the pi-calculus, by preserving the object paradigms. To preserve the object notation, they name the pi-calculus processes with the name of the objects. The consistency between sequence diagrams and statechart diagrams can be checked by verifying that the messages in the sequence diagrams trigger states in statechart diagrams.

Padmanabhan Krishnan [KR00] describes a framework in which UML diagrams can be formalized to perform consistency checking. UML diagrams are translated into specifications of the theorem proving tool PVS (Prototype Verification System). The PVS is a language that allows for the introduction of abstract data types, functions etc. To check for consistency between sequence and class diagrams, the class diagrams must first be annotated with OCL constraints. The PVS will check if the sequence of states described in the sequence diagram can be obtained from the class diagrams. Custom traces (i.e. sequence of states) can also be supplied by the user to check if other properties hold.

Soon-Kyeong Kim and David Carrington [KI04] describe how consistency checking between different UML models can be accomplished by using a formal object-oriented metamodeling approach. They formally define the abstract syntax and semantics of the UML model using Object-Z as a metalanguage. They then define consistency constraints that logically exist between semantically equivalent elements in the metamodel but are not defined in the current UML metamodel structure. Once the consistency constraints have been defined for each of the UML model elements, consistency checking between different model elements can be achieved by verifying that the combined models preserve all of the consistency constraints for the individual model elements. They use the formal language to ensure the consistency between two diagrams. However, in my approach I do not use formal language and I ensure the consistency between multiple statechart diagrams and sequence diagrams.

CHAPTER 3 - SUPER STATE ANALYSIS (*SSA*) APPROACH

3.1 The Super State

My approach for consistency analysis combines the state information of multiple state diagrams into a composite super state, SS. The super state has the form [s_1, s_2, …, s_n] where s_i is the state of object *i* and n is the total number of objects. A system may have many different super states depending on the number of objects that are being analyzed. The super state details all of the possible composite states the objects can be in as well as the transition pairs which lead from one composite state to another. In this way the super state provides the complete collaborative view of a set of objects in the model.

Super State may change after each message call. For every call we have <SS_{pre}, call, SS_{post}> where SS_{pre} is the super state before call and SS_{post} is the super state after the message call has been called. In SS_{post}, only the state of one object may change. This object must be the destination object of the message call. The state of the other objects remains in the same state as they were before the call. We calculate the super state of multiple state diagrams after each valid transition and that is used to evaluate each sequence diagram. A sequence diagram to be valid should be a subsequence of the set of sequences that are possible in a super state. Invalid and impossible sequences can be identified.

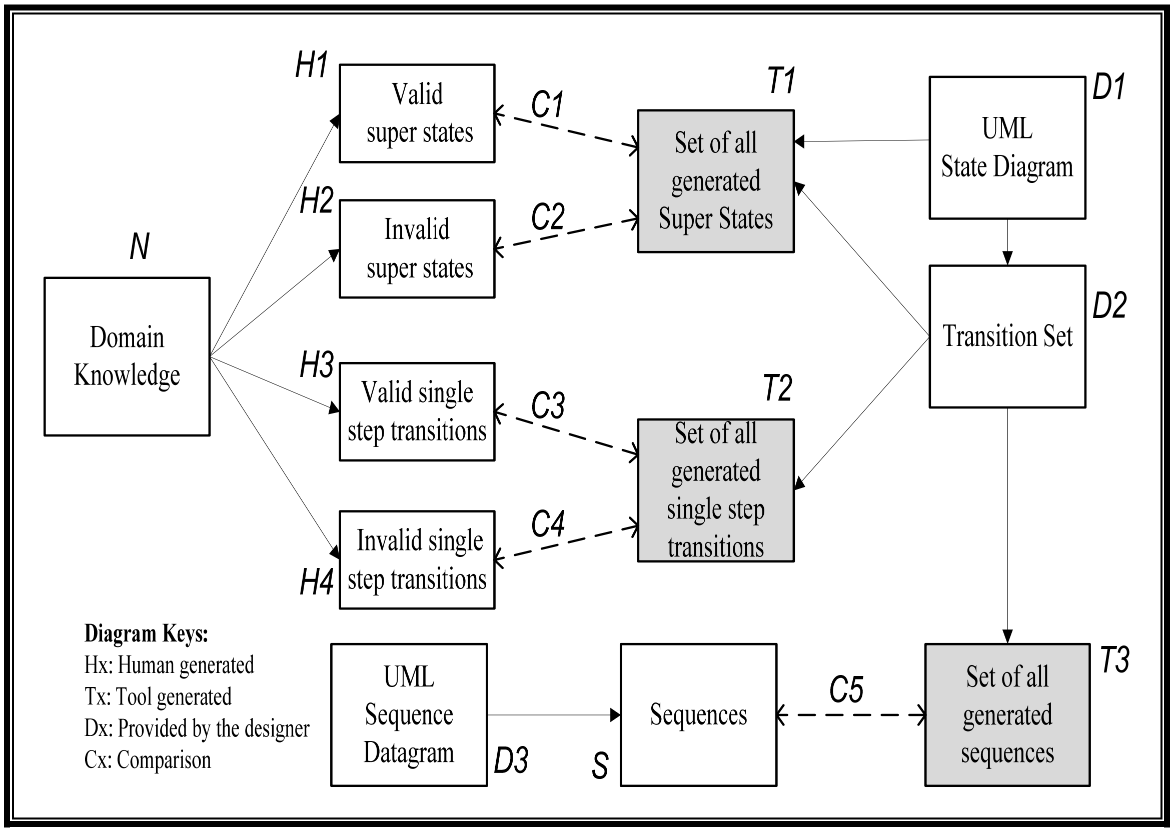
3.2 Super State Analysis

The information in UML diagrams are related to each other and represent different views of a system. Hence, they can be validated against each other. Given a statechart diagram, researchers [LI03] have shown how to validate it against a sequence diagram. On the other hand, given a sequence diagram, it can be validated against a statechart diagram [DU00, SH06].

However, I am proposing a new approach to check the consistency between multiple state diagrams and one or more sequence diagrams. My analysis, the *Super State Analysis* (SSA), focuses on multiple state diagrams instead of a single state diagram.

The diagram on Figure 3.1 shows the complete analysis process and the relationships between the different sources of information. Some information is known from the domain knowledge and provided by the developer while some other information is extracted from the existing information and generated automatically. *Super State Analysis* uses the provided information to generate some information automatically. Comparing the information from different sources allows us to detect the inconsistencies. *SSA* includes some inconsistencies that can be detected by the computer and some other faults that can be identified by the human. *Super State Analysis* performs five types of comparisons to detect the inconsistencies.

Figure 3.1 *SSA* Model



The *SSA* model on Figure 3.1 includes the 12 information sets that are involved in *Super State Analysis*. The system developer provides the UML state diagrams, the transition set and UML sequence diagrams (D1, D2, and D3). The developer identifies the valid super states, invalid super states, valid single step transitions, and the invalid single step transitions (H1, H2, H3, and H4). *SSA* is automatically generates three large sets: set of all generated super states, set of all generated single step transitions, and set of all generated sequences (T1, T2, and T3). These sets are generated using the UML state diagrams and the provided transition set. The valid sequences (S) are extracted from the UML sequence diagram. Table 3.1 describes each component involved in the analysis and the source of each.

Table 3.1 Description of each component involved in *SSA* Model

Box	Name	Description	Source
N	Domain Knowledge	The facts that are known by the developer of the system	Known from the domain knowledge
H1	Valid Super States	The set of states that are identified to be valid super states.	Domain Knowledge
H2	Invalid Super States	The set of states that are identified to be invalid super states.	Domain Knowledge
H3	Valid single step transitions	The set of transitions that are identified to be valid single step transitions	Domain Knowledge
H4	Invalid single step transitions	The set of transitions that are identified to be invalid single step transitions	Domain Knowledge
T1	Set of all generated Super States	These super states are generated automatically using the UML diagram and transition set	Generated Automatically by *SSA*
T2	Set of all single step transition	This set contains all of the single step transitions. These transitions are generated automatically using the transition set	Generated Automatically by *SSA*
T3	Set of all generated sequences	This set contains all of the legal sequences that are allowed by the system. This set is generated automatically using the transition set	Generated Automatically by *SSA*
D1	UML State Diagram	The state diagrams that are written by the developer who specifies the system	Developer
D2	Transition Set	The set of all legal transitions that are allowed by the system. The developer provides this set	Developer
D3	UML Sequence Datagram	The sequence diagrams that are written by the developer who specifies the system	Developer
S	Sequences	Sequences that are extracted from the UML sequence diagrams	Generated Automatically by *SSA*

Super State Analysis uses the UML state diagram (D1) and the transition set (D2) to generate the set of all generated Super States (T1). Also, *SSA* uses the transition set (D2) to compute the set of all generated sequences (T3). Moreover, *SSA* uses the transition set to compute the set of all generated single step transitions (T2). The developer uses the domain knowledge to identify the valid super states, invalid super states, valid single step transitions, and invalid single step transitions. Furthermore, the UML sequence diagram is used to extract the sequences which will be compared to the set of all generated sequences.

3.3 Comparisons

The Super State Analysis consists of five types of comparisons to detect the inconsistencies in the multiple state diagrams and sequence diagrams.

1. C1: Compares the set of all generated super states (T1) with the set of valid super states (H1).
2. C2: Compares the set of all generated super states (T1) with the set of invalid super states (H2).
3. C3: Compares the set of all generated single step transitions (T2) with the set of valid single step transitions (H3).
4. C4: Compares the set of all generated single step transitions (T2) with the set of invalid single step transitions (H4).
5. C5: Compares the set of all generated sequences (T3) with the set of sequences (S) which are extracted from the provided UML sequence diagrams.

C1 and C2 detect the valid and invalid super states while C3 and C4 identify the valid and invalid single step transitions. C5 detects the invalid sequences. This comparison is fully automated since both T3 and S are generated automatically. The other four comparisons can be automated if we formalize the four sets: H1, H2, H3, and H4 and feed them to the system. By comparing these four sets to the generated sets: T1 and T2 the inconsistencies can be detected automatically.

3.4 The Transition Matrix

The transition matrix details the possible global states of the system based on a vector of states of individual instances of classes and the possible transitions between the states in the super state (SS). Consider a program that has class X and class Y. Let class X has an initial state A and two other states, B and C, while class Y has an initial state D and a second state E. Figure 3.2 shows the state diagram for class X and Figure 3.3 shows the state diagram for class Y. The state diagrams depict how instances of X and Y can transition between those states. Let class Y makes the transition between state D and state E whenever class X makes the transition from state A to state B. Table 3.2 shows possible transitions in the super state that is the cross-product of all states with one instance of X and one instance of Y.

Figure 3.2 State Diagram for Class X

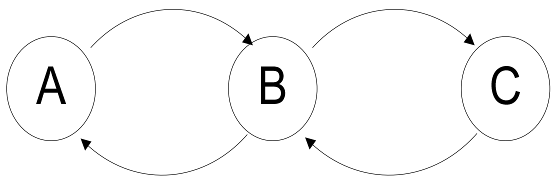
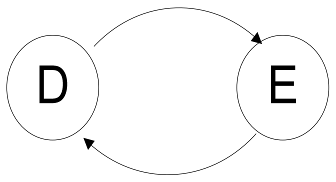


Figure 3.3 State Diagram for Class Y



An entry in a cell in T_1 (Table 3.2) shows that in one step, the system can transition from the state of the row to the state of the column. Taking the product of T_1

by itself gives a matrix that contains the transitions possible with two steps. The closure of T_1 is the sum of products, $T_1 + T_1*T_1 + T_1*T_1*T_1 +\dots$ The closure shows all possible transitions in any number of steps. Although the closure is represented as an infinite sum, it can be calculated in at most the number of products equal to the rank of the initial matrix. In most cases, it is even smaller than that number.

Table 3.2 Super state transition matrix T_1

T_1	AD	BD	CD	AE	BE	CE
AD	0	0	0	0	1	0
BD	1	0	1	0	0	0
CD	0	1	0	0	0	0
AE	0	1	0	0	0	0
BE	0	0	0	1	0	1
CE	0	0	0	0	1	0

3.5 The Transition Set

There is some essential information about the relationships between transitions in different state diagrams that is not captured in any UML diagram. This information includes the fact that some transitions are paired. This information is critical to understanding the specified system because the state of one class could affect the state of another class. Also, identifying the paired relations is important when building the system to maintain the consistency between the state diagrams. These relations between states of different state diagrams help the system to identify which states are paired and hence maintain the consistency. Looking to just a single state diagram without considering the others could leave the system in an inconsistent state.

3.5.1 Transition Set Types

In the transition set, there are three types of transitions: *independent transitions*, *paired transitions*, and *constrained transitions*. The independent transitions are the transitions that can happen individually without influencing states and transitions of other state diagrams. The effect of those transitions is local within their state diagrams and they do not consider the state of other diagrams. They may change only the state of the diagrams that they are belongs to.

The paired transitions are those transitions that must happen together. If a transition is paired to other transition(s), then they must happen simultaneously. The effect of those transitions is global since they enforce other transition(s) to happen and hence may change the super state.

The constrained transitions are the transitions that can happen only when some other state diagrams are in specific states. The state of other diagrams may prevent the constrained transition. This kind of transitions considers the state of other diagrams. Our interest is the paired and constrained transitions since they interact with multiple state diagrams.

3.5.2 Example

Consider a simple ATM system that has two state diagrams: customer state diagram (Figure 3.4) and account state diagram (Figure 3.5). The customer will be in good standing (G) until an overdraft transaction is happened then the customer will go to state N (NotGoodStanding). The account stays in P (Positive) until a withdrawal transaction happened with amount that exceed the available balance in which case the account will became negative (V). We labeled the transitions in Figure 3.4 and Figure 3.5 for ease of reference.

Figure 3.4 State Diagram for Customer

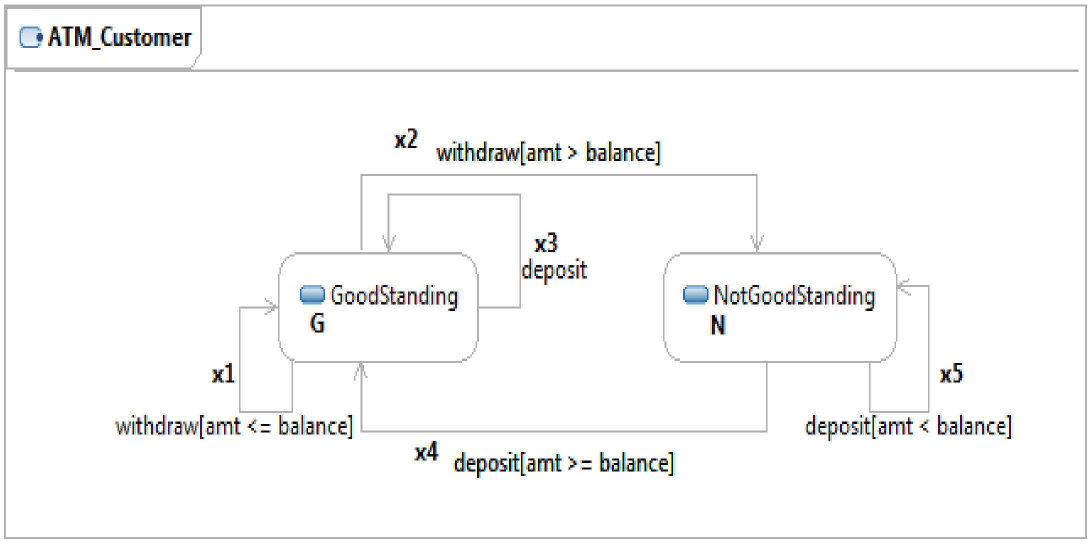
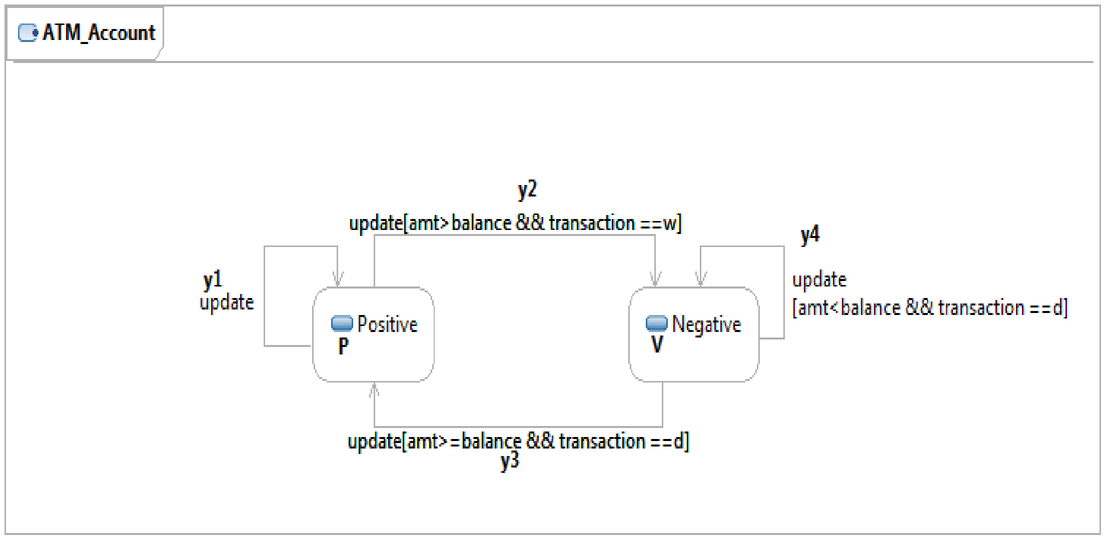


Figure 3.5 State Diagram for Account


ATM_Account
y2
update[amt>balance && transaction ==w]
y4
update
[amt<balance && transaction ==d]
y1
update
Positive
P
Negative
V
update[amt>=balance && transaction ==d]
y3

The proposed transition set technique links the transitions of multiple state diagrams together to capture the relationship information of the paired transitions. The transition set includes explicitly all legal transitions that are allowed in the system. This set links transitions of multiple state diagrams together. The transition set allows viewing the super state (global state) of the system rather than individual state of a single object.

The complete information that is in the transition set is not stated explicitly in any UML diagram. Partial information could be inferred from the set of correct sequence diagrams. In order to have the complete information inferred from the sequence diagrams, we must have all possible correct sequence diagrams. Having the explicit transition set is easier and more realistic than inferring them from sequence diagrams.

An entry in the transition set has the form [PreState, (transitions), PostState] where PreState is the super state before transitions and PostState is the super state after the transitions taken. The transitions has the form (t_1, t_2, …, t_n) where t_i are the paired transitions. i.e. must happen together.

In the transition set of the ATM example, we have the following entries

[GP, (x1, y1), GP]
[GP, (x3, y1), GP]
[GP, (x2, y2), NV]
[NV, (x5, y4), NV]
[NV, (x4, y3), GP]

If we don’t consider the transition set, the system can make some illegal transitions. For example, [GP, x2, NP] or [NV, y3, NP]. Having the correct transition set provided for the system will prevent such inconsistencies.

3.6 Inconsistency Detection

Super State Analysis (*SSA*) discovers inconsistencies in super states, single step transitions, and sequences.

3.6.1 State Inconsistencies

The valid and invalid states will possibly be identified by *SSA*. If a super state (SS) is generated by Box T1, but it is not in the set of valid states (Box H1) then the state is an invalid SS. This could happen if there is a wrong transition in the transition set. On the other hand, if a super state is in the set of valid states (Box H1), but it is not generated by Box T1, then this SS is a valid super state and should be generated. SS wouldn’t be generated if there is a missing transition in the transition set or in the state diagram.

The following kinds of inconsistencies can be discovered by this analysis:

i. Valid super states

ii. Invalid super states

3.6.2 Single Step Transitions Inconsistencies

The valid and invalid single step transitions (Box H3 and Box H4) are known from the domain knowledge. The set of all generated single step transitions (Box T2) are generated automatically using the transition set. Comparing those sets will discover some legal and illegal transitions.

If a valid transition does not appear in the set of all generated single step transitions that means this transition is missing. Furthermore, if an invalid transition appears in the set of all generated single step transitions that mean this transition is illegal.

The following kinds of inconsistencies are discovered by this analysis:

i. Valid single step transitions

ii. Invalid single step transitions

3.6.3 Sequence Inconsistencies

Super State Analysis generates the sequences using the transition matrix. To validate a UML sequence diagram, *SSA* extracts the sequences first (Box S), then, compares them to the set of all generated sequences (Box T3). If there is a matching sequence in that set, this sequence is valid. Otherwise, it is an invalid sequence.

The following kinds of inconsistencies are discovered by this analysis:

i. Illegal sequences

Super State Analysis uses the UML state diagrams and the transition set to generate the set of all generated Super States (SS). Also, *SSA* uses the transition set to compute the set of all generated sequences. Moreover, *SSA* uses the transition set to compute the set of all generated single step transitions.

From the domain knowledge, we identify the sets of valid and invalid Super States (SS) and the valid and invalid single step transitions. The UML sequence diagram is used to extract the sequences which will be compared to the set of all generated sequences.

The inconsistency can be fixed by several ways. It can be fixed by adding or removing a fact to the domain knowledge. Another way to fix the inconsistencies is correcting the state diagram by adding a new transition (or removing one).

CHAPTER 4 - CASE STUDY I (LIBRARY EXAMPLE)

4.1 Description

This case study describes the interaction between a patron of a library and the copies of books the library holds. In order to simplify the model the library holds only one copy of each book. Figure 4.1 shows the class diagram for this model. Figure 4.2 and Figure 4.3 are the state diagrams for the patron and book objects. Note that the transitions in the state diagrams are numbered for ease of reference. This example originally was created by a team of students trying to create a correct model of a simple library system.

The patron object can be in one of three states: *Good Standing, Too Many Books,* and *Fines*. We will call these states *G, T,* and *F* respectively for the rest of this chapter. A patron starts in *G* until the number of books the patron has checked out is equal to *MAX* or the patron returns an overdue book. In the former, the patron will transition to state *T* where they will remain until they return a book. In the latter, the patron will transition to *F* where they will not be able to do anything until they pay the fine that is owed.

A book object has six states: *On Shelf, Missing, On Hold, Checked Out, Overdue*, and *Returned*. We will call these states *O, M, H, C, D,* and *R* respectively for the rest of this chapter.

The two transitions from *C* labeled *check* represent the library determining if the book is overdue. If the book is overdue it will transition to *D*. Otherwise, it will transition to *R* where it will remain until the library places it back on the shelf.

Figure 4.1 Class Diagram for the library example

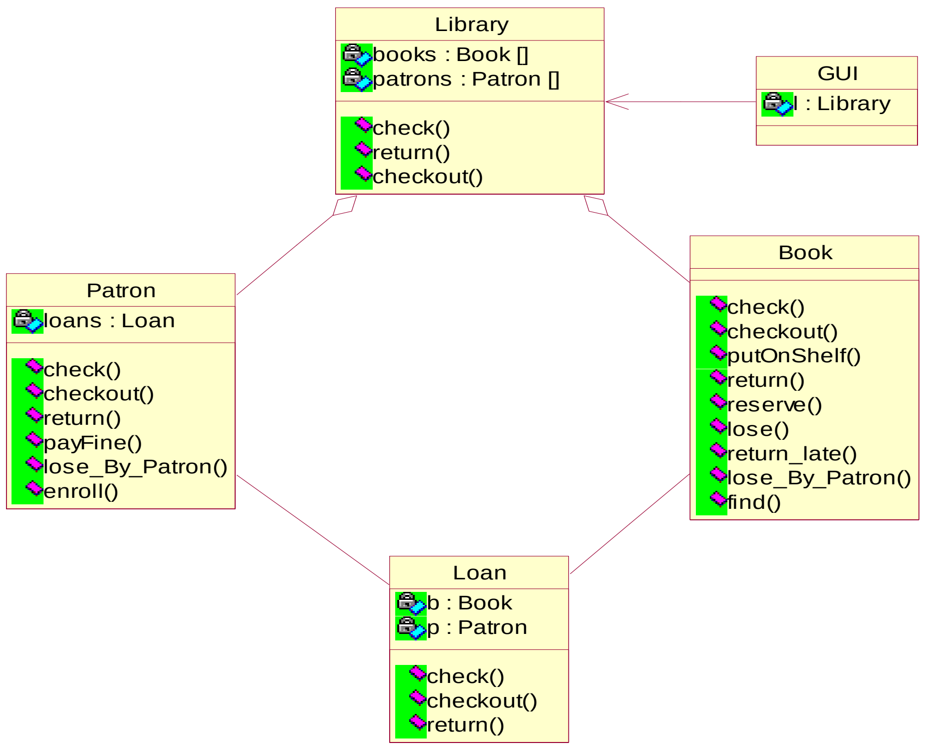


Figure 4.2 State Diagram for Patron

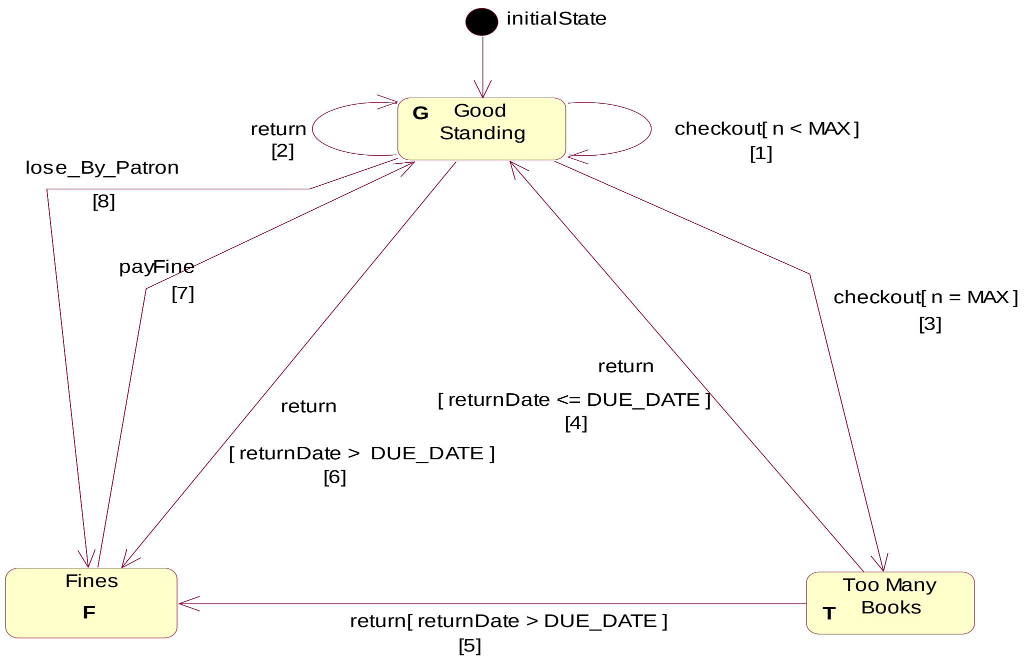
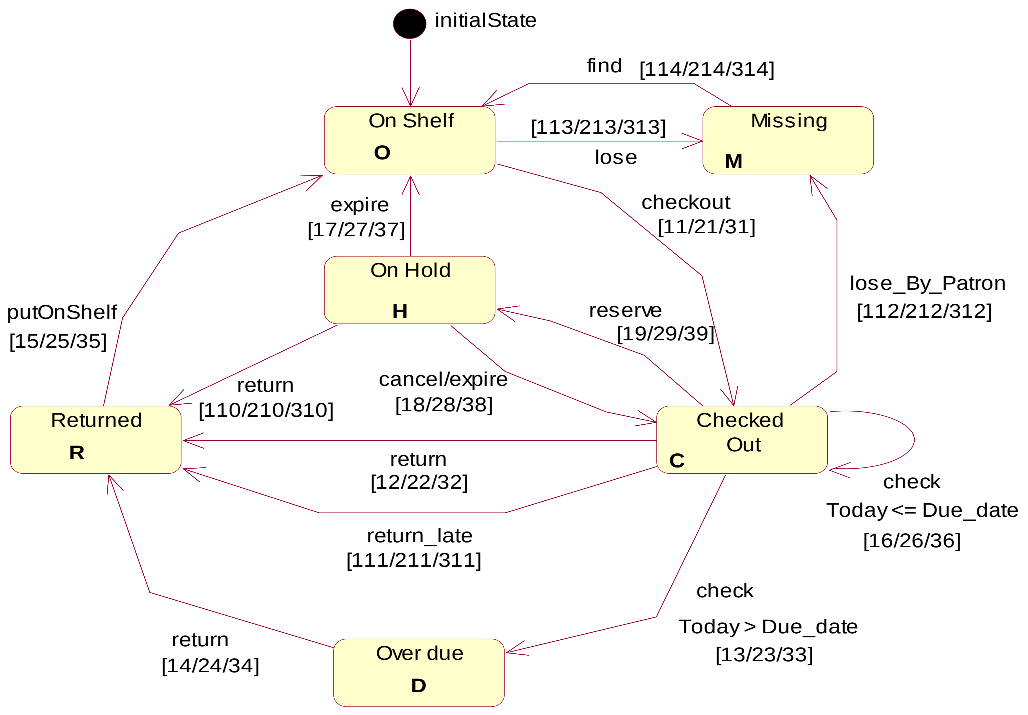


Figure 4.3 State Diagram for Book



4.2 The Library example invariant

1. The system starts with the initial super state *SS* where the patron is in *G* and the Book is in *O*.
2. The patron can check out a book only if she/he is in *G* state.
3. The patron should always be able to return a book at any time.
4. When the number of books checked out by Patron is equal to *MAX*, the state of patron should be changed from *G* to *T*.
5. When the number of books checked out by Patron is not equal to *MAX*, the state of patron should not be in *T*.
6. The patron should be able to return a missing book at any time.

7. The number of *n* for a patron is increased by 1 when the patron checks out or reserves a book.
8. The number of *n* for a patron is decreased by 1 when the patron returns a book.
9. *n* is set to 0 when the system starts.
10. When a book is lost by a patron, the state of that patron should change to *F*.
11. The Patron cannot be in *T* and at least the state of one book is in *O* or *R*.
12. If the patron loses one book, she/he cannot lose another one until the fine is paid first.
13. If the patron loses one book, she/he cannot return another one until the fine is paid first.
14. If the patron returns one book late, she/he cannot lose another one (until she/he pay the fine).
15. The patron can check out and return books even if the other books are on over due

4.3 Analysis

For our analysis we will assume that the library has only one patron and three books. We now pair the transitions from the patron and book objects that can occur together. An 'X' indicates that we are not concerned about the state of the object. The transition set is shown in Table 4.2.

The initial transition matrix A_1 has column and row headings with quadruple representing the states of the four objects. For this model there are 3*6*6*6 = 648 combinations of the four objects. Table 4.1 shows a portion of the initial transition matrix A_1.

Table 4.1 Portion of A_1

A_1	GOOO	GOCO	GODO	GORO	GCOO
GOOO		1,21			1,11
GOCO		26	23	2,22	
GODO					
GORO	25				
GCOO					16

The row headings are the initial states and the column headings are the final states. The numbers in the table arise from Figure 4.2 and Figure 4.3. For the purpose of clarification we have assigned unique numeric identifiers to the transitions for each instance of an object in our system. The book object has three numeric identifiers for each transition since we have three instances of that object.

For example, $GOOO \rightarrow GOCO$ represents a patron in good standing checking out the second book. The 1 indicates the patron took the transition labeled *checkout* [$n < MAX$] and the 21 indicates the second book took the transition labeled *checkout*. If there is an entry for a cell in the matrix then the transition is valid. A_2 is defined as $A_1 * A_1$ which identifies all the states we can reach in two steps. Table 4.3 shows a portion of A_2.

Table 4.2 Transition set for Library Example

$SS_{pre} \rightarrow SS_{post}$	Transition	Description
GOXX → GCXX	checkout[n<MAX], checkout	Check out a book (if at least one X = O || R)
GXOX → GXCX	checkout[n<MAX], checkout	
GXXO → GXXC	checkout[n<MAX], checkout	
GOXX → TCXX	checkout[n=MAX], checkout	Check out a book (if X = C || H || D)
GXOX → TXCX	checkout[n=MAX], checkout	
GXXO → TXXC	checkout[n=MAX], checkout	
GCXX → GRXX	return, return	Return book on time
GXCX → GXRX	return, return	
GXXC → GXXR	return, return	
GDXX → FRXX	return[returnDate>dueDate], return	Return an over due book
GXDX → FXRX	return[returnDate>dueDate], return	
GXXD → FXXR	return[returnDate>dueDate], return	
TCXX → GRXX	return[returnDate<=dueDate], return	Patron with MAX books returns a book on time
TXCX → GXRX	return[returnDate<=dueDate], return	
TXXC → GXXR	return[returnDate<=dueDate], return	
TDXX → FRXX	return[returnDate>dueDate], return	Patron with MAX books returns an over due book
TXDX → FXRX	return[returnDate>dueDate], return	
TXXD → FXXR	return[returnDate>dueDate], return	
GCXX → FMXX	lose_by_patron, lose_by_patron	Patron lost a book
GXCX → FXMX	lose_by_patron, lose_by_patron	
GXXC → FXXM	lose_by_patron, lose_by_patron	
GCXX → GHXX	reserve	Patron holds a book
GXCX → GXHX	reserve	
GXXC → GXXH	reserve	
TCXX → THXX	reserve	Patron with MAX books holds a book
TXCX → TXHX	reserve	
TXXC → TXXH	reserve	
GHXX → GCXX	cancel/expire	Cancel/Expiration of holding book (n < MAX)
GXHX → GXCX	cancel/expire	
GXXH → GXXC	cancel/expire	
THXX → TCXX	cancel/expire	Cancel/Expiration of holding book (n = MAX)
TXHX → TXCX	cancel/expire	
TXXH → TXXC	cancel/expire	
O → M	lose	A book lost by the library
M → O	find	A book found by the library
F → G	payFine	Patron pays fine
C → D	check[today>Due_date]	Book becomes over due
C → C	check[today<=Due_date]	Book remains checked out
R → O	putOnShelf	Book is re-shelved
H→ R	return	Return an on hold book
C→ R	Return_late	Return a late book

Table 4.3 Portion of A_2

A_2	GOOO	GOCO	GODO
GOOO		(1,21)(26)	(1,21)(23)
GOCO	(2,22)(25)	(26)(26)	(26)(23)
GODO			
GORO		(25)(1,21)	
GCOO	(2,12)(15)		

From Table 4.3 we can observe that it is possible to go from *GOCO* to *GOOO* by first returning the second book and then shelving it.

For this model, the invalid states include two sets. The first set includes the states where the patron is in *T* and one of the three books is in *O* or *R*. Clearly the patron cannot have *MAX* books checked out if one of the books is not checked out. The other set of invalid states occurs when the patron is in *F* and all books are in *C* or *D*. In order for the patron to be in *F,* one of the three books would have had to have been returned. An analysis of A^* for this model shows that the columns for these invalid states are empty.

Some of the faults in the design of the library example can be discovered by simply analyzing the transition matrix. One such fault was a missing transition. From *FRCO* and *FCRO* there is no valid single step transition to *FRRO*. This means that if one book is returned late, the patron goes to *F* status and cannot return the other book until the fine is paid.

4.4 Some inconsistencies found in the library example

1. The patron cannot return the book if she/he find it later on.

   GCXX, lose_by_patron, FMXX, ? , GRXX

   This can be fixed by adding the following paired transitions:

   (*F,find_by _patron,G*) on Patron state diagram and (*M,find_by_parton,R*) on

Book state diagram

2. The patron cannot return any of her/his other books until the fine is paid first.

   GDDX, return(late), FRDX, ?, FRRX or GCCX, return(late), FRCX, ?, FRRX

   This can be fixed by adding (*F, return, F*) on Patron state diagram and pair it with *(C, return, R)* and *(D, return, R)* on Book state diagram

   In general, the patron cannot do anything if she/he in on 'F' until she/he pays the fine.

3. The patron cannot lose an over due book

   GCXX, check, GDXX, ?, FMXX

   This can be fixed by adding *(D, lose_by_patron, M*) on Book state diagram and pair it with *(G, lose_by_patron, F)* on Patron state diagram

4. The patron cannot lose a book if he is in state 'T'

   TCXX, ?, FMXX

   This can be fixed by adding *(T, lose_by_patron, F*) on Patron state diagram and pair it with *(C, lose_by_patron, M)* on Book state diagram

5. The system reaches an invalid state when the patron checked out MAX books and trying to return a book late. TRXX , TXRX, and TXXR are invalid states because the patron cannot be in state T while one of the his books is returned.

Example: GOOO(checkout[n<MAX],checkout)→ GCOO(checkout[n<MAX],checkout) → GCCO (checkout[n=MAX],checkout)→ TCCC(return_late) → TRCC

This can be fixed by paring transition *return_late* in Book with transition *return[returnDate>DUEDATE]* in Parton

CHAPTER 5 - CASE STUDY II (UNIVERSITY EXAMPLE)

5.1 Description

The case study in this chapter describes a university system. The university consists of colleges where each college may have students, instructors, and courses. The students can enroll to section of courses. The instructors teach section of courses. Figure 5.1 shows the class diagram for the university model. In this case study, we will study the behavior (states) of 6 classes in the university model. Specificity, the state diagrams of the following classes will be considered: enrollment, teaching, student, instructor, section, and room. Figure 5.2, Figure 5.3, Figure 5.4, Figure 5.5, Figure 5.6, and Figure 5.7 show the state diagrams for each class. Statistical information about the University Model is shown in Table 5.1.

Table 5.1 Information about the University Model

Number of classes	12	
Number of State Diagrams	6	
State Diagram	Number of states	Number of transitions
Enrollment	13	20
Teaching	4	5
Student	6	13
Instructor	4	12
Section	3	5
Room	3	6

Figure 5.1 Class Diagram for Univeristy Model

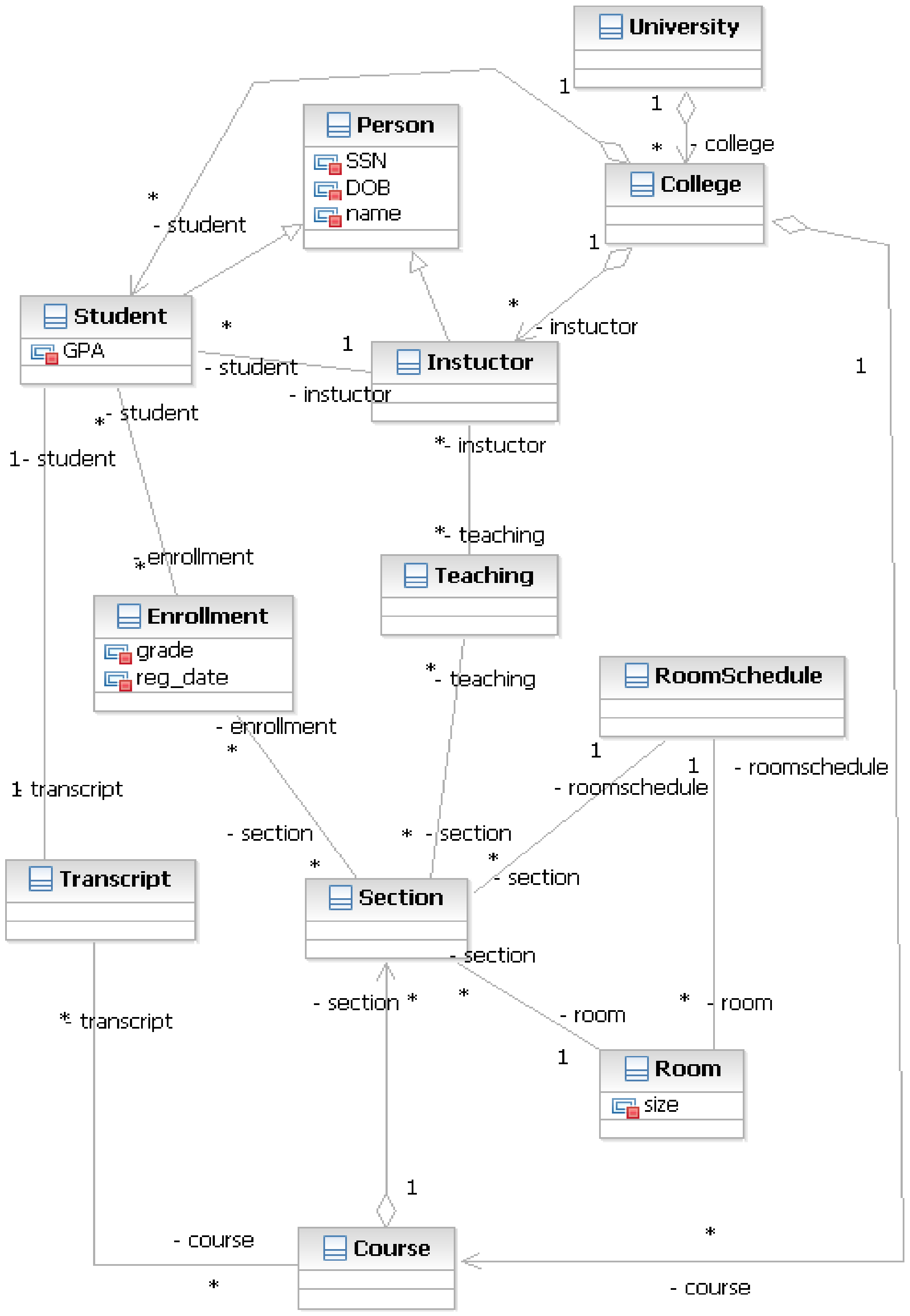


Figure 5.2 State Diagram for Enrollment

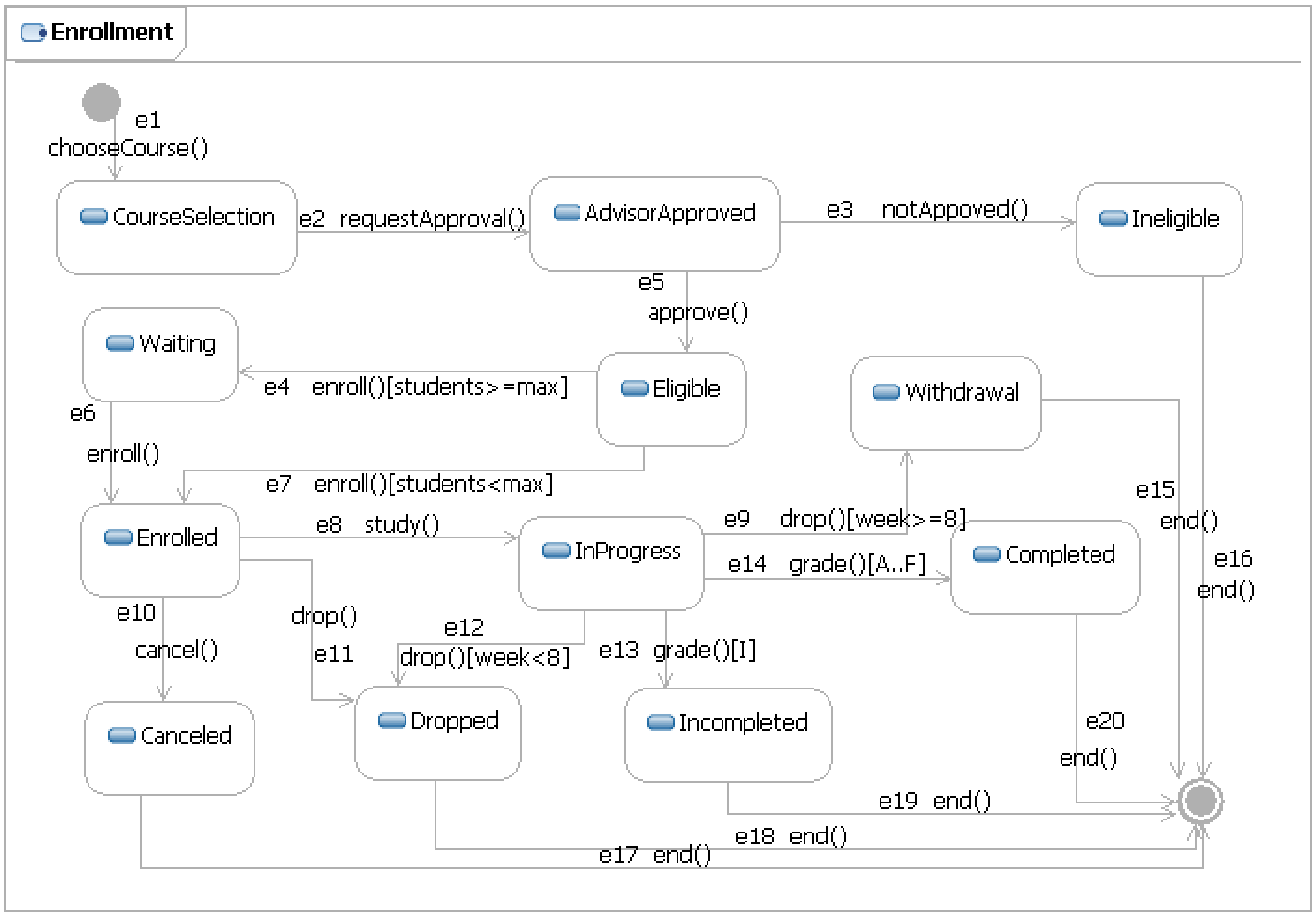


Figure 5.3 State Diagram for Teaching

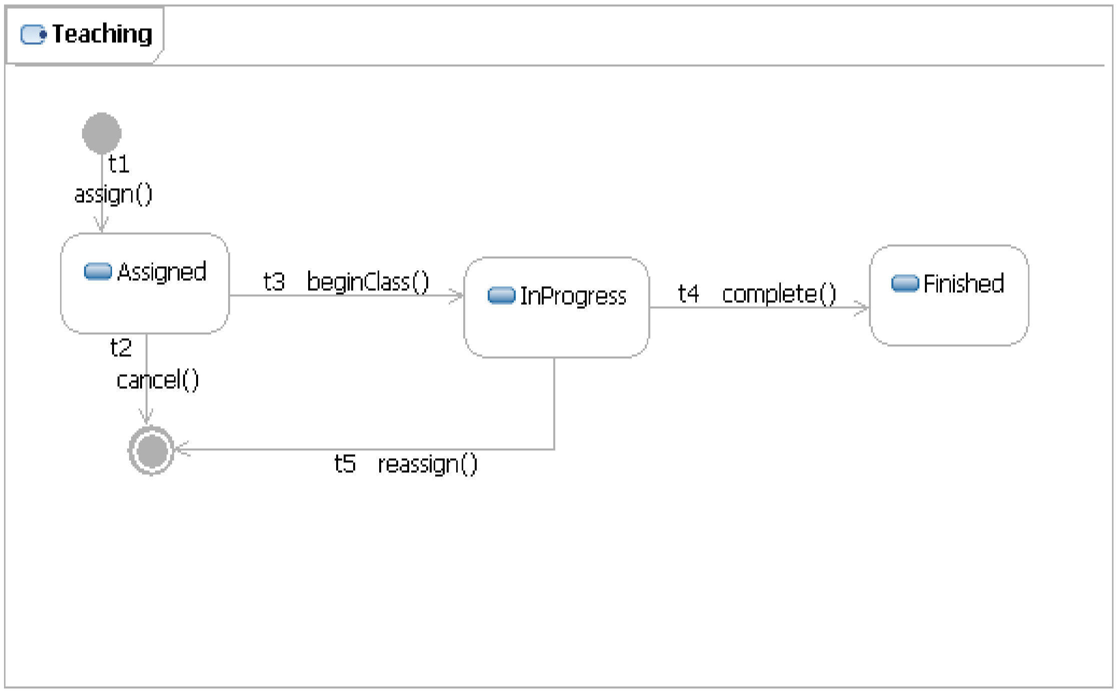


Figure 5.4 State Diagram for Student

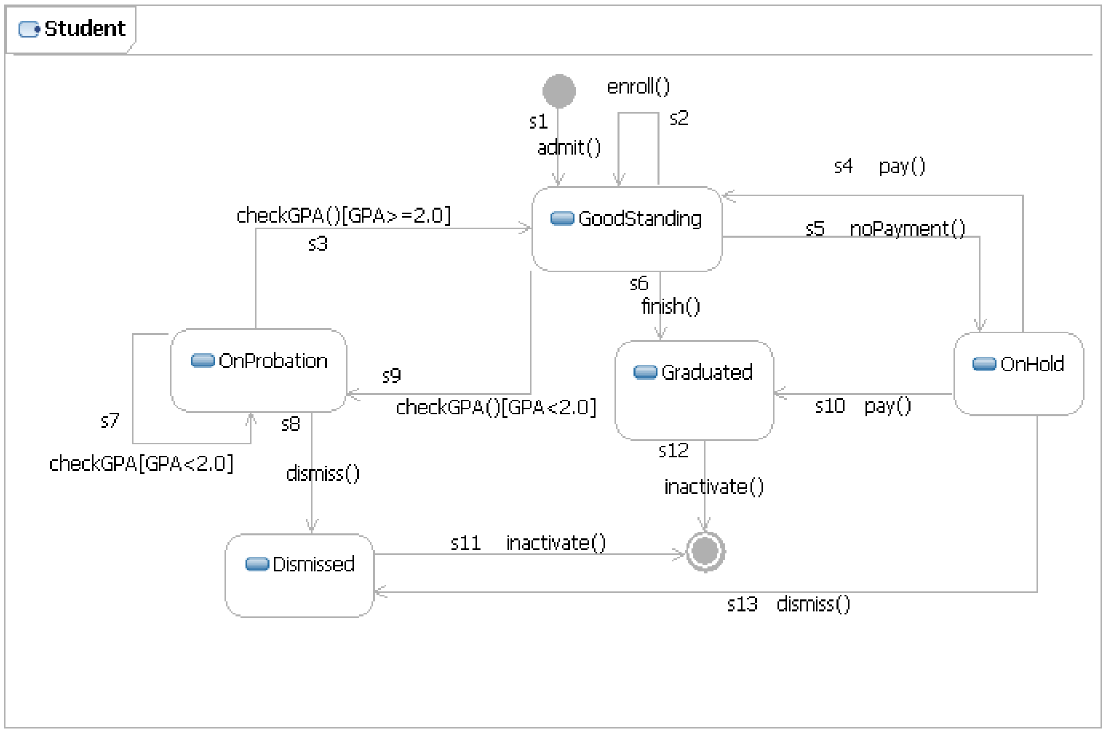


Figure 5.5 State Diagram for Instructor

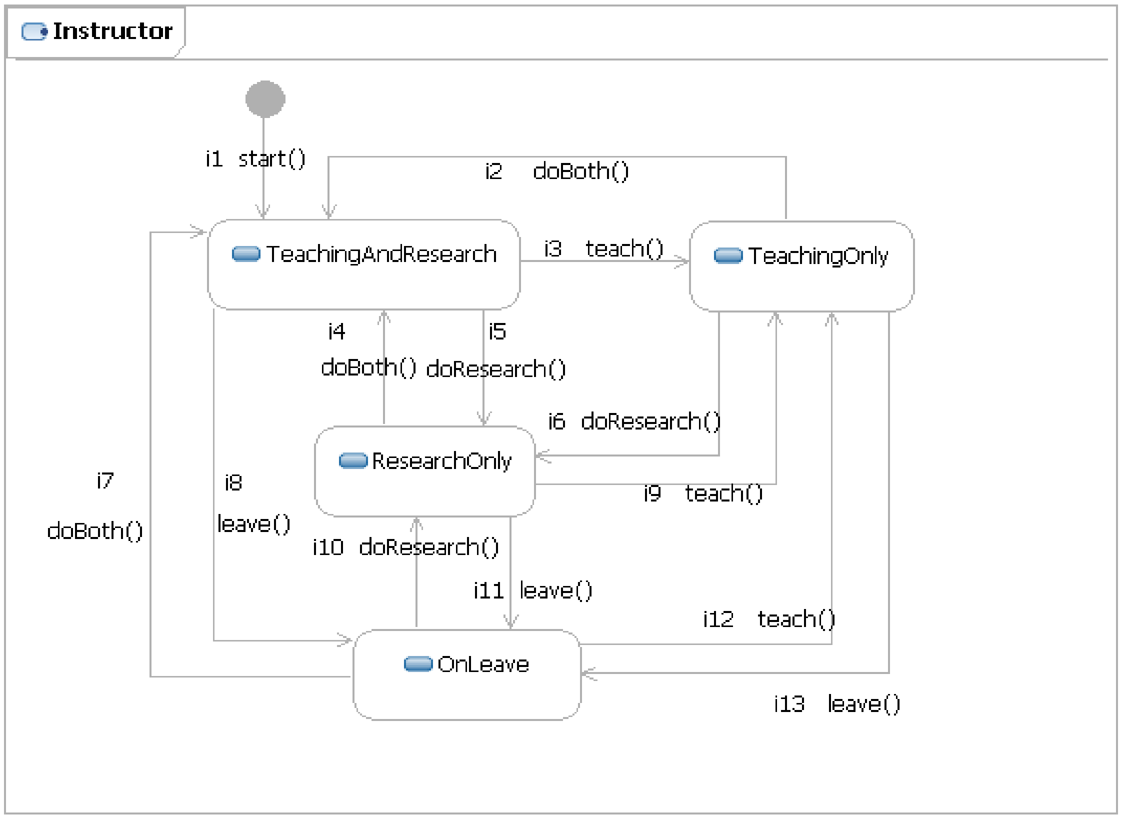


Figure 5.6 State Diagram for Section

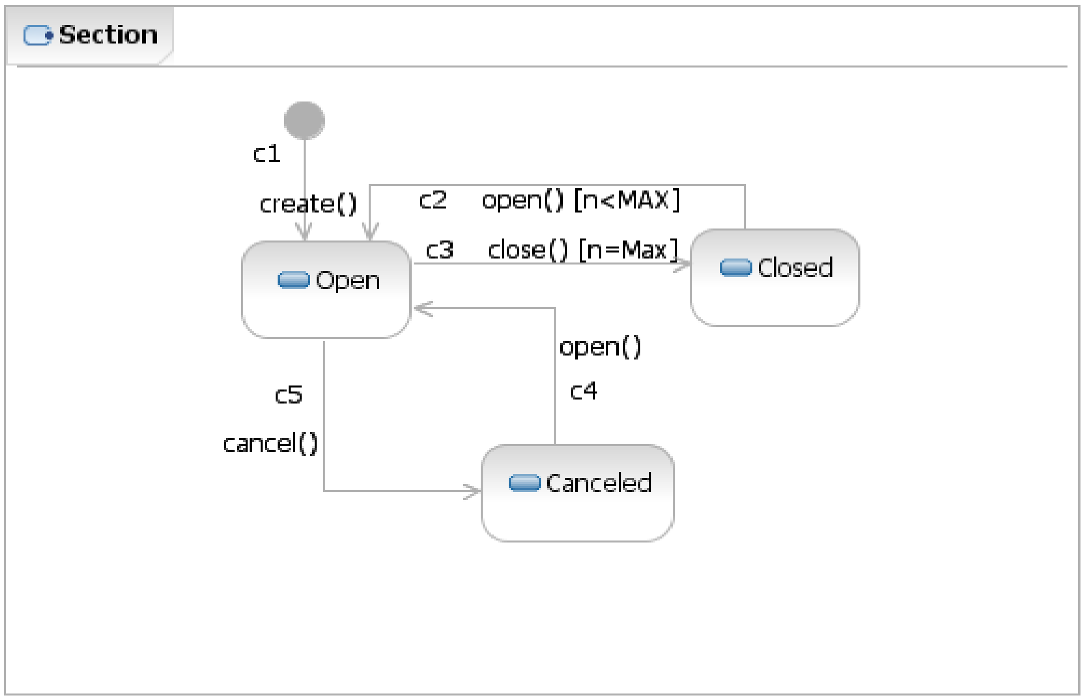
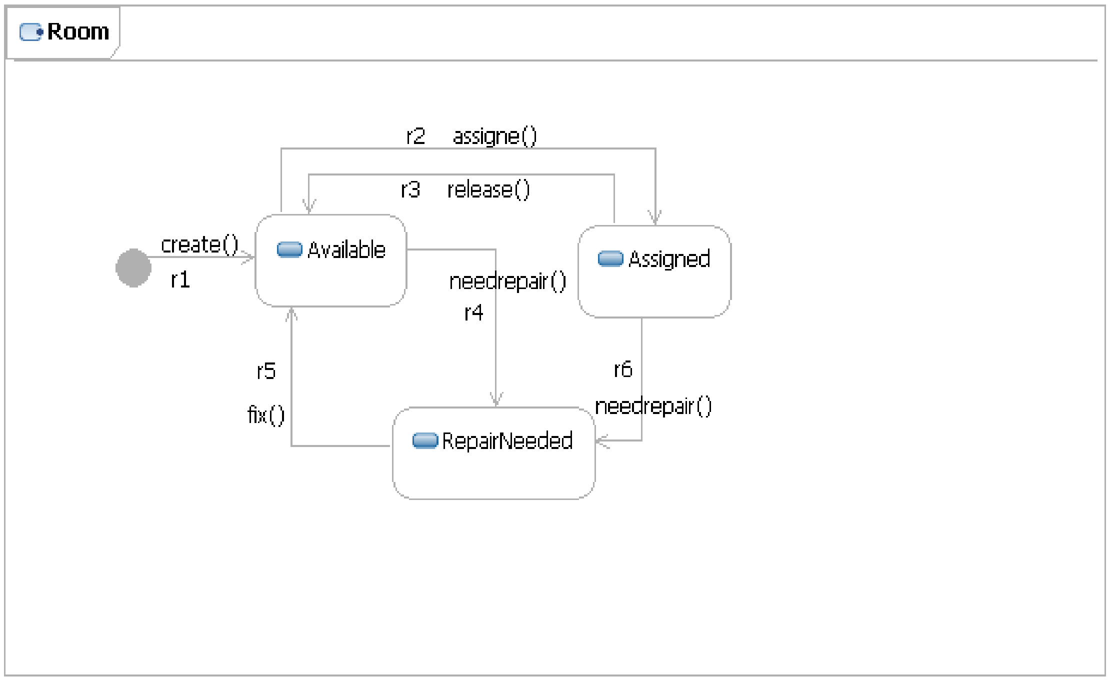


Figure 5.7 State Diagram for Room



5.2 The State Diagrams for University Model (UM)

The enrollment object can be in one of the following states: *CourseSelection, AdvisorApproval, Ineligible, Waiting, Eligible, Withdrawal, Enrolled, InProgress, Completed, Cancelled, Dropped,* and *Incomplete.* We will call these states *C, A, I, W, E, T, L, P, M, K, D*, and *N* respectively for the rest of this chapter.

A teaching object has four states *Assigned, InProgress, Finished,* and *End*. We will call these states *A, P, F,* and *Z* respectively for the rest of this chapter.

The student object can be in one of the following states: *GoodStanding, OnHold, Graduated, OnProbation, Dismissed,* and *End.* We will call these states *G, H, R, P, D,* and *Z* respectively for the rest of this chapter.

An instructor object has four states *TeachingAndResearch, TeachingOnly, ResearchOnly,* and *OnLeave*. We will call these states *B, T, R,* and *L* respectively for the rest of this chapter.

The section object can be in one of the following states: *Open, Closed,* and *Canceled.* We will call these states *O, C,* and *N* respectively for the rest of this chapter.

A room object has three states *Available, Assigned*, and *RepairNeeded*. We will call these states *A, S and R* respectively for the rest of this chapter.

5.3 Some Invariants for UM

1. The system starts with the initial super state SS where the student in Good Standing, instructor in both TeachingAndResearch, enrollment in Course Selection, teaching in assigned, section in opened, and room in available.
   Student= G, Instructor=B, Enrollment=C, Teaching=A, Section=O, Room=A
2. The student can enroll in classes only if s/he is in good standing.
3. The student can enroll in a section only if the section is open.
4. The teaching for an instructor can be assigned only if the instructor is on teaching only or in TeachingAndResearch.
5. The teaching begins only if the room is assigned.
6. The teaching begins only if the section is not canceled.
7. If an instructor go to on leave or research only after the class is begin the teaching must be reassigned.
8. The student graduated when all his/her classes are completed.
9. When a section is canceled, the enrollment is canceled too.

10. If a student tries to enroll in a closed section due to the capacity, the enrollment of that student should be placed on waiting until the section is opened.

5.4 Analysis

For our analysis we will study the behavior of the university model in two cases. The first case is when the system has one object of *student*, one object of *section*, one object of *enrollment*, one object of *instructor*, one object of *teaching*, and one object of *room*. The transition set for this case is shown in Table 5.2. In this case the total number of possible states:

6*3*13*4*4*3 = 11232 possible states

In the second case we will study the behavior when the system has two objects of *student*, two objects of *section*, two objects of *enrollment*, two objects of *instructor*, two object of *teaching*, and two objects of *room*. The transition set for this case is shown in Table 5.3. In this case the total number of possible states:

6*6*3*3*13*13*4*4*4*4*3*3 = 126157824 possible states

In the transition sets in Table 5.2 and Table 5.3, we pair the transitions from the objects that can occur together. An ‘X’ indicates that we are not concerned about the state of object.

Each super state in Table 5.2 consists of six states. The SS has the form (S_1, S_2, S_3, S_4, S_5, S_6) where S_1 is the state of *student,* S_2 is the state of *enrollment,* S_3 is the state of *section,* S_4 is the state of *room,* S_5 is the state of *instructor,* and S_6 is the state of *teaching*.

Table 5.2 Transition Set for University Model

For a system with one student, one section, one instructor, and one room
S= student, E=enrollment, C=section, R=room, I=instructor, T=teaching

	SS_{pre} S,E,C,R,I,T		SS_{post} S,E,C,R,I,T	Transition(s)
1	G,X,X,X,X,X	→	H,X,X,X,X,X	nopayment
2	H,X,X,X,X,X	→	G,X,X,X,X,X	Pay
3	G,X,X,X,X,X	→	R,X,X,X,X,X	Finish
4	H,X,X,X,X,X	→	R,X,X,X,X,X	Pay
5	H,X,X,X,X,X	→	D,X,X,X,X,X	dismiss
6	G,X,X,X,X,X	→	P,X,X,X,X,X	checkGPA[GPA<2.0]
7	P,X,X,X,X,X	→	P,X,X,X,X,X	checkGPA[GPA<2.0]
8	P,X,X,X,X,X	→	G,X,X,X,X,X	checkGPA[GPA>=2.0]
9	P,X,X,X,X,X	→	D,X,X,X,X,X	dismiss
10	R,X,X,X,X,X	→	Z,X,X,X,X,X	inactivate
11	D,X,X,X,X,X	→	Z,X,X,X,X,X	inactivate
12	X,X,X,X,B,X	→	X,X,X,X,T,X	teach
13	X,X,X,X,T,X	→	X,X,X,X,B,X	doBoth
14	X,X,X,X,B,X	→	X,X,X,X,R,X	doresearch
15	X,X,X,X,R,X	→	X,X,X,X,B,X	doBoth
16	X,X,X,X,B,X	→	X,X,X,X,L,X	leave
17	X,X,X,X,L,X	→	X,X,X,X,B,X	doBoth
18	X,X,X,X,T,X	→	X,X,X,X,L,X	leave
19	X,X,X,X,L,X	→	X,X,X,X,T,X	teach
20	X,X,X,X,R,X	→	X,X,X,X,L,X	leave
21	X,X,X,X,L,X	→	X,X,X,X,R,X	doResearch
22	X,X,X,X,T,X	→	X,X,X,X,R,X	doResearch
23	X,X,X,X,R,X	→	X,X,X,X,T,X	teach
24	G,E,O,X,X,X	→	G,L,O,X,X,X	enroll, enroll[students<max]
25	G,E,O,X,X,X	→	G,L,C,X,X,X	enroll,enroll[students<max], close[n<max]
26	G,E,C,X,X,X	→	G,W,C,X,X,X	enroll, enroll[students>=max]
27	G,C,X,X,X,X	→	G,A,X,X,X,X	requestApproval

28	G,A,X,X,X,X	→	G,I,X,X,X,X	notApproved
29	G,A,X,X,X,X	→	G,E,X,X,X,X	approve
30	G,W,C,X,X,X	→	G,L,O,X,X,X	enroll, enroll, open[n<max]
31	G,L,X,S,B,P	→	G,P,X,S,B,P	study
32	G,L,X,S,T,P	→	G,P,X,S,T,P	study
33	G,L,X,X,X,X	→	G,D,X,X,X,X	drop
34	G,L,O,X,X,X	→	G,K,N,X,X,X	cancel, cancel
35	G,P,X,X,X,X	→	G,D,X,X,X,X	drop [week<8]
36	G,P,X,X,X,X	→	G,N,X,X,X,X	grade[I]
37	G,P,X,X,X,X	→	G,M,X,X,X,X	grade[A..F]
38	G,P,X,X,X,X	→	G,T,X,X,X,X	drop[week>=8]
39	G,I,X,X,X,X	→	G,Z,X,X,X,X	end
40	G,T,X,X,X,X	→	G,Z,X,X,X,X	end
41	G,M,X,X,X,X	→	G,Z,X,X,X,X	end
42	G,N,X,X,X,X	→	G,Z,X,X,X,X	end
43	G,D,X,X,X,X	→	G,Z,X,X,X,X	end
44	G,K,X,X,X,X	→	G,Z,X,X,X,X	end
45	X,X,X,X,X,A	→	X,X,X,X,X,P	beginClass
46	X,X,X,X,X,P	→	X,X,X,X,X,Z	reassign
47	X,X,X,X,X,A	→	X,X,X,X,X,Z	cancel
48	X,X,X,X,X,P	→	X,X,X,X,X,F	complete
49	X,X,O,X,X,X	→	X,X,C,X,X,X	close[max=n]
50	X,X,O,X,X,X	→	X,X,N,X,X,X	cancel
51	X,X,C,X,X,X	→	X,X,O,X,X,X	open[n<max]
52	X,X,N,X,X,X	→	X,X,O,X,X,X	open
53	X,X,X,A,X,X	→	X,X,X,S,X,X	assign
54	X,X,X,A,X,X	→	X,X,X,R,X,X	needrepair
55	X,X,X,S,X,X	→	X,X,X,R,X,X	needrepair
56	X,X,X,S,X,X	→	X,X,X,A,X,X	release
57	X,X,X,R,X,X	→	X,X,X,A,X,X	fix

Each super state in Table 5.3 consists of twelve states. The SS has the form (S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10}, S_{11}, S_{12}) where S_1 is the state of first *student,* S_2 is the state of second *student,* S_3 is the state of first *enrollment,* S_4 is the state of second *enrollment,* S_5 is the state of first *section,* S_6 is the state of second *section,* S_7 is the state of first *room,* S_8 is the state of second *room,* S_9 is the state of first *instructor,* S_{10} is the state of second *instructor,* S_{11} is the state of first *teaching*, and S_{12} is the state of second *teaching*.

Table 5.3 Transition Set for University Model

For a system with 2 students, 2 sections, 2 instructors, and 2 rooms
s= student, e=enrollment, c=section, r=room, i=instructor, t=teaching

	SS_{pre}		SS_{post}	Transition(s)
#	s1e1c1r1i1t1s2e2c2r2i2t2	→	s1e1c1r1i1t1s2e2c2r2i2t2	
1	G,X,X,X,X,X,X,X,X,X,X,X	→	H,X,X,X,X,X,X,X,X,X,X,X	nopayment
2	H,X,X,X,X,X,X,X,X,X,X,X	→	G,X,X,X,X,X,X,X,X,X,X,X	Pay
3	G,X,X,X,X,X,X,X,X,X,X,X	→	R,X,X,X,X,X,X,X,X,X,X,X	Finish
4	H,X,X,X,X,X,X,X,X,X,X,X	→	R,X,X,X,X,X,X,X,X,X,X,X	Pay
5	H,X,X,X,X,X,X,X,X,X,X,X	→	D,X,X,X,X,X,X,X,X,X,X,X	dismiss
6	G,X,X,X,X,X,X,X,X,X,X,X	→	P,X,X,X,X,X,X,X,X,X,X,X	checkGPA[GPA<2.0]
7	P,X,X,X,X,X,X,X,X,X,X,X	→	P,X,X,X,X,X,X,X,X,X,X,X	checkGPA[GPA<2.0]
8	P,X,X,X,X,X,X,X,X,X,X,X	→	G,X,X,X,X,X,X,X,X,X,X,X	checkGPA[GPA>=2.0]
9	P,X,X,X,X,X,X,X,X,X,X,X	→	D,X,X,X,X,X,X,X,X,X,X,X	dismiss
10	R,X,X,X,X,X,X,X,X,X,X,X	→	Z,X,X,X,X,X,X,X,X,X,X,X	inactivate
11	D,X,X,X,X,X,X,X,X,X,X,X	→	Z,X,X,X,X,X,X,X,X,X,X,X	inactivate
12	X,X,X,X,X,X,G,X,X,X,X,X	→	X,X,X,X,X,X,H,X,X,X,X,X	nopayment
13	X,X,X,X,X,X,H,X,X,X,X,X	→	X,X,X,X,X,X,G,X,X,X,X,X	Pay
14	X,X,X,X,X,X,G,X,X,X,X,X	→	X,X,X,X,X,X,R,X,X,X,X,X	Finish
15	X,X,X,X,X,X,H,X,X,X,X,X	→	X,X,X,X,X,X,R,X,X,X,X,X	Pay
16	X,X,X,X,X,X,H,X,X,X,X,X	→	X,X,X,X,X,X,D,X,X,X,X,X	dismiss
17	X,X,X,X,X,X,G,X,X,X,X,X	→	X,X,X,X,X,X,P,X,X,X,X,X	checkGPA[GPA<2.0]
18	X,X,X,X,X,X,P,X,X,X,X,X	→	X,X,X,X,X,X,P,X,X,X,X,X	checkGPA[GPA<2.0]

19	X,X,X,X,X,X,P,X,X,X,X,X → X,X,X,X,X,X,G,X,X,X,X,X	checkGPA[GPA>=2.0]
20	X,X,X,X,X,X,P,X,X,X,X,X → X,X,X,X,X,X,D,X,X,X,X,X	dismiss
21	X,X,X,X,X,X,R,X,X,X,X,X → X,X,X,X,X,X,Z,X,X,X,X,X	inactivate
22	X,X,X,X,X,X,D,X,X,X,X,X → X,X,X,X,X,X,Z,X,X,X,X,X	inactivate
23	X,X,X,X,B,X,X,X,X,X,X,X → X,X,X,X,T,X,X,X,X,X,X,X	teach
24	X,X,X,X,T,X,X,X,X,X,X,X → X,X,X,X,B,X,X,X,X,X,X,X	doBoth
25	X,X,X,X,B,X,X,X,X,X,X,X → X,X,X,X,R,X,X,X,X,X,X,X	doresearch
26	X,X,X,X,R,X,X,X,X,X,X,X → X,X,X,X,B,X,X,X,X,X,X,X	doBoth
27	X,X,X,X,B,X,X,X,X,X,X,X → X,X,X,X,L,X,X,X,X,X,X,X	leave
28	X,X,X,X,L,X,X,X,X,X,X,X → X,X,X,X,B,X,X,X,X,X,X,X	doBoth
29	X,X,X,X,T,X,X,X,X,X,X,X → X,X,X,X,L,X,X,X,X,X,X,X	leave
30	X,X,X,X,L,X,X,X,X,X,X,X → X,X,X,X,T,X,X,X,X,X,X,X	teach
31	X,X,X,X,R,X,X,X,X,X,X,X → X,X,X,X,L,X,X,X,X,X,X,X	leave
32	X,X,X,X,L,X,X,X,X,X,X,X → X,X,X,X,R,X,X,X,X,X,X,X	doResearch
33	X,X,X,X,T,X,X,X,X,X,X,X → X,X,X,X,R,X,X,X,X,X,X,X	doResearch
34	X,X,X,X,R,X,X,X,X,X,X,X → X,X,X,X,T,X,X,X,X,X,X,X	teach
35	X,X,X,X,X,X,X,X,X,X,B,X → X,X,X,X,X,X,X,X,X,X,T,X	teach
36	X,X,X,X,X,X,X,X,X,X,T,X → X,X,X,X,X,X,X,X,X,X,B,X	doBoth
37	X,X,X,X,X,X,X,X,X,X,B,X → X,X,X,X,X,X,X,X,X,X,R,X	doresearch
38	X,X,X,X,X,X,X,X,X,X,R,X → X,X,X,X,X,X,X,X,X,X,B,X	doBoth
39	X,X,X,X,X,X,X,X,X,X,B,X → X,X,X,X,X,X,X,X,X,X,L,X	leave
40	X,X,X,X,X,X,X,X,X,X,L,X → X,X,X,X,X,X,X,X,X,X,B,X	doBoth
41	X,X,X,X,X,X,X,X,X,X,T,X → X,X,X,X,X,X,X,X,X,X,L,X	leave
42	X,X,X,X,X,X,X,X,X,X,L,X → X,X,X,X,X,X,X,X,X,X,T,X	teach
43	X,X,X,X,X,X,X,X,X,X,R,X → X,X,X,X,X,X,X,X,X,X,L,X	leave
44	X,X,X,X,X,X,X,X,X,X,L,X → X,X,X,X,X,X,X,X,X,X,R,X	doResearch
45	X,X,X,X,X,X,X,X,X,X,T,X → X,X,X,X,X,X,X,X,X,X,R,X	doResearch
46	X,X,X,X,X,X,X,X,X,X,R,X → X,X,X,X,X,X,X,X,X,X,T,X	teach
47	G,E,O,X,X,X,X,X,X,X,X,X → G,L,O,X,X,X,X,X,X,X,X,X	enroll, enroll[students<max]
48	G,E,O,X,X,X,X,X,X,X,X,X → G,L,C,X,X,X,X,X,X,X,X,X	enroll,enroll[students< max], close [n=max]

49	G,E,C,X,X,X,X,X,X,X,X,X	→	G,W,C,X,X,X,X,X,X,X,X,X	enroll, enroll[students>=max]
50	G,C,X,X,X,X,X,X,X,X,X,X	→	G,A,X,X,X,X,X,X,X,X,X,X	requestApproval
51	G,A,X,X,X,X,X,X,X,X,X,X	→	G,I,X,X,X,X,X,X,X,X,X,X	notApproved
52	G,A,X,X,X,X,X,X,X,X,X,X	→	G,E,X,X,X,X,X,X,X,X,X,X	approve
53	G,W,C,X,X,X,X,X,X,X,X,X	→	G,L,O,X,X,X,X,X,X,X,X,X	enroll, enroll, open[n<max]
54	G,L,X,S,B,P,X,X,X,X,X,X	→	G,P,X,S,B,P,X,X,X,X,X,X	study
55	G,L,X,S,T,P,X,X,X,X,X,X	→	G,P,X,S,T,P,X,X,X,X,X,X	study
56	G,L,X,X,X,X,X,X,X,X,X,X	→	G,D,X,X,X,X,X,X,X,X,X,X	drop
57	G,L,O,X,X,X,X,X,X,X,X,X	→	G,K,N,X,X,X,X,X,X,X,X,X	cancel, cancel
58	G,P,X,X,X,X,X,X,X,X,X,X	→	G,D,X,X,X,X,X,X,X,X,X,X	drop [week<8]
59	G,P,X,X,X,X,X,X,X,X,X,X	→	G,N,X,X,X,X,X,X,X,X,X,X	grade[I]
60	G,P,X,X,X,X,X,X,X,X,X,X	→	G,M,X,X,X,X,X,X,X,X,X,X	grade[A..F]
61	G,P,X,X,X,X,X,X,X,X,X,X	→	G,T,X,X,X,X,X,X,X,X,X,X	drop[week>=8]
62	G,I,X,X,X,X,X,X,X,X,X,X	→	G,Z,X,X,X,X,X,X,X,X,X,X	end
63	G,T,X,X,X,X,X,X,X,X,X,X	→	G,Z,X,X,X,X,X,X,X,X,X,X	end
64	G,M,X,X,X,X,X,X,X,X,X,X	→	G,Z,X,X,X,X,X,X,X,X,X,X	end
65	G,N,X,X,X,X,X,X,X,X,X,X	→	G,Z,X,X,X,X,X,X,X,X,X,X	end
66	G,D,X,X,X,X,X,X,X,X,X,X	→	G,Z,X,X,X,X,X,X,X,X,X,X	end
67	G,K,X,X,X,X,X,X,X,X,X,X	→	G,Z,X,X,X,X,X,X,X,X,X,X	end
68	X,X,X,X,X,X,G,E,O,X,X,X	→	X,X,X,X,X,X,G,L,O,X,X,X	enroll, enroll[students<max]
69	X,X,X,X,X,X,G,E,O,X,X,X	→	X,X,X,X,X,X,G,L,C,X,X,X	enroll, enroll[students< max], close[n=max]
70	X,X,X,X,X,X,G,E,C,X,X,X	→	X,X,X,X,X,X,G,W,C,X,X,X	enroll, enroll[students>=max]
71	X,X,X,X,X,X,G,C,X,X,X,X	→	X,X,X,X,X,X,G,A,X,X,X,X	requestApproval
72	X,X,X,X,X,X,G,A,X,X,X,X	→	X,X,X,X,X,X,G,I,X,X,X,X	notApproved
73	X,X,X,X,X,X,G,A,X,X,X,X	→	X,X,X,X,X,X,G,E,X,X,X,X	approve
74	X,X,X,X,X,X,G,W,C,X,X,X	→	X,X,X,X,X,X,G,L,O,X,X,X	enroll, enroll, open[n<max]
75	X,X,X,X,X,X,G,L,X,S,B,P	→	X,X,X,X,X,X,G,P,X,S,B,P	study
76	X,X,X,X,X,X,G,L,X,S,T,P	→	X,X,X,X,X,X,G,P,X,S,T,P	study
77	X,X,X,X,X,X,G,L,X,X,X,X	→	X,X,X,X,X,X,G,D,X,X,X,X	drop

78	X,X,X,X,X,X,G,L,O,X,X,X → X,X,X,X,X,X,G,K,N,X,X,X	cancel, cancel
79	X,X,X,X,X,X,G,P,X,X,X,X → X,X,X,X,X,X,G,D,X,X,X,X	drop [week<8]
80	X,X,X,X,X,X,G,P,X,X,X,X → X,X,X,X,X,X,G,N,X,X,X,X	grade[I]
81	X,X,X,X,X,X,G,P,X,X,X,X → X,X,X,X,X,X,G,M,X,X,X,X	grade[A..F]
82	X,X,X,X,X,X,G,P,X,X,X,X → X,X,X,X,X,X,G,T,X,X,X,X	drop[week>=8]
83	X,X,X,X,X,X,G,I,X,X,X,X → X,X,X,X,X,X,G,Z,X,X,X,X	end
84	X,X,X,X,X,X,G,T,X,X,X,X → X,X,X,X,X,X,G,Z,X,X,X,X	end
85	X,X,X,X,X,X,G,M,X,X,X,X → X,X,X,X,X,X,G,Z,X,X,X,X	end
86	X,X,X,X,X,X,G,N,X,X,X,X → X,X,X,X,X,X,G,Z,X,X,X,X	end
87	X,X,X,X,X,X,G,D,X,X,X,X → X,X,X,X,X,X,G,Z,X,X,X,X	end
88	X,X,X,X,X,X,G,K,X,X,X,X → X,X,X,X,X,X,G,Z,X,X,X,X	end
89	X,X,X,X,X,A,X,X,X,X,X,X → X,X,X,X,X,P,X,X,X,X,X,X	beginClass
90	X,X,X,X,X,P,X,X,X,X,X,X → X,X,X,X,X,Z,X,X,X,X,X,X	reassign
91	X,X,X,X,X,A,X,X,X,X,X,X → X,X,X,X,X,Z,X,X,X,X,X,X	cancel
92	X,X,X,X,X,P,X,X,X,X,X,X → X,X,X,X,X,F,X,X,X,X,X,X	complete
93	X,X,X,X,X,X,X,X,X,X,X,A → X,X,X,X,X,X,X,X,X,X,X,P	beginClass
94	X,X,X,X,X,X,X,X,X,X,X,P → X,X,X,X,X,X,X,X,X,X,X,Z	reassign
95	X,X,X,X,X,X,X,X,X,X,X,A → X,X,X,X,X,X,X,X,X,X,X,Z	cancel
96	X,X,X,X,X,X,X,X,X,X,X,P → X,X,X,X,X,X,X,X,X,X,X,F	complete
97	X,X,O,X,X,X,X,X,X,X,X,X → X,X,C,X,X,X,X,X,X,X,X,X	close[max=n]
98	X,X,O,X,X,X,X,X,X,X,X,X → X,X,N,X,X,X,X,X,X,X,X,X	cancel
99	X,X,C,X,X,X,X,X,X,X,X,X → X,X,O,X,X,X,X,X,X,X,X,X	open[n<max]
100	X,X,N,X,X,X,X,X,X,X,X,X → X,X,O,X,X,X,X,X,X,X,X,X	open
101	X,X,X,X,X,X,X,X,O,X,X,X → X,X,X,X,X,X,X,X,C,X,X,X	close[max=n]
102	X,X,X,X,X,X,X,X,O,X,X,X → X,X,X,X,X,X,X,X,N,X,X,X	cancel
103	X,X,X,X,X,X,X,X,C,X,X,X → X,X,X,X,X,X,X,X,O,X,X,X	open[n<max]
104	X,X,X,X,X,X,X,X,N,X,X,X → X,X,X,X,X,X,X,X,O,X,X,X	open
105	X,X,X,A,X,X,X,X,X,X,X,X → X,X,X,S,X,X,X,X,X,X,X,X	assign
106	X,X,X,A,X,X,X,X,X,X,X,X → X,X,X,R,X,X,X,X,X,X,X,X	needrepair
107	X,X,X,S,X,X,X,X,X,X,X,X → X,X,X,R,X,X,X,X,X,X,X,X	needrepair
108	X,X,X,S,X,X,X,X,X,X,X,X → X,X,X,A,X,X,X,X,X,X,X,X	release
109	X,X,X,R,X,X,X,X,X,X,X,X → X,X,X,A,X,X,X,X,X,X,X,X	fix
110	X,X,X,X,X,X,X,X,X,A,X,X → X,X,X,X,X,X,X,X,X,S,X,X	assign

111	X,X,X,X,X,X,X,X,X,A,X,X	$\rightarrow$	X,X,X,X,X,X,X,X,X,R,X,X	needrepair
112	X,X,X,X,X,X,X,X,X,S,X,X	$\rightarrow$	X,X,X,X,X,X,X,X,X,R,X,X	needrepair
113	X,X,X,X,X,X,X,X,X,S,X,X	$\rightarrow$	X,X,X,X,X,X,X,X,X,A,X,X	release
114	X,X,X,X,X,X,X,X,X,R,X,X	$\rightarrow$	X,X,X,X,X,X,X,X,X,A,X,X	fix

The initial transition matrix A_1 has column and row headings with *Super States* representing the state of each of the six objects. Table 5.4 shows portion of the initial transition matrix A_1 for the University Model. The row headings are the initial states and the column headings are the final states. The identifiers in the table arise from Figure 5.2 - Figure 5.7.

For the purpose of clarification we have assigned unique identifiers to the transitions for each object in the University Model. Each transition is denoted by a letter and a number. The letter refers to the object's name and the number refers to the transition number within the object. For example, (GEOABA$\rightarrow$ GLOABA) represents an eligible student in good standing enrolls in a course. The *e7* indicates the enrollment took the transition labeled *enroll[students<max]* and *s2* indicates the student took the transition labeled *enroll*. If there is an entry for a cell in the matrix then the transition is valid. If the cell is empty then there is no transition can lead from the initial SS to the final SS. For example (GCOABA$\rightarrow$ GIOABA), there is no way to go from GCOABA to GCOABA in one step.

Table 5.4 Portion of A_1 for the transition set in Table 5.2

A_1	GCOABA	GAOABA	GIOABA	GWOABA	GEOABA	GTOABA	GLOABA	GPOABA	GMOABA	GKOABA	GDOABA	GNOABA	GZOABA
GCOABA		e2											
GAOABA			e3		e5								
GIOABA													e16
GWOABA							e6						
GEOABA				e4			e7,s2						
GTOABA													e15
GLOABA								e8		e10	e11		
GPOABA						e9			e14		e12	e13	
GMOABA													e20
GKOABA													e17
GDOABA													e18
GNOABA													e19
GZOABA													

Table 5.5 shows a portion of A_2. Any SS that can be reached in two steps are shown in A_2. From Table 5.5 we can observe that it is possible to go from GAOABA to GLOABA by:

- first go from GAOABA to GEOABA by doing *e5*
- then go from GEOABA to GLOABA by doing *e7* and *s2*

Also, we can go from GCOABA to GIOABA by:

- first go from GCOABA to GAOABA by doing *e2*
- then go from GAOABA to GIOABA by doing *e3*

Table 5.5 Portion of A_2 for the transition set in Table 5.2

A_2	GCOABA	GAOABA	GIOABA	GWOABA	GEOABA	GTOABA	GLOABA	GPOABA	GMOABA	GKOABA	GDOABA	GNOABA	GZOABA
GCOABA			e2, e3										
GAOABA				e5, e4			e5, (e7,s2)						e3,e16
GIOABA													
GWOABA								e6,e8					
GEOABA							e4,e6	(e7,s2), e8		(e7,s2), e10	(e7,s2), e11		
GTOABA													
GLOABA						e8, e9			e8, e14		e8,e12	e8, e13	e10,e17 || e11,e18
GPOABA													e9,e15 || e14,e20 || e12,e18 || e13,e19
GMOABA													
GKOABA													
GDOABA													
GNOABA													
GZOABA													

If a cell has more than one entry, it means that there are more than one path can lead from the initial state to the final state. For example, the system can go from (GPOABA) to (GZOABA) in two steps by several ways: *e9* then *e15* or *e14* then *e20* or *e12* then *e18* or *e13* then *e19*.

By looking to the transition set, we may figure out that there are three types of transitions: independent transitions, paired transitions, and constrained transitions. Those types of transitions were described earlier in chapter 3.

In the transition set of the university model (Table 5.2), we may classify the transitions according to the above types as the following:

1. Transitions 1-23 and 45-57 are independent transitions.
2. Transitions 24, 25, 26, 30, and 34 are paired transitions.
3. Transitions 24-44 are constrained transitions.

Most of the transitions in Table 5.2 are independent transitions (36 out of 57). The independent transition can happen at any time without considering state of the other objects. Furthermore, the independent transitions do not change the state of other objects. They can only change the state of objects that they are belongs to. For instance, the student can go from *GoodStanding* (G) to *OnHold* (H) by doing the transition *noPayment* (G,X,X,X,X,X → H,X,X,X,X,X) regardless of the state of the other objects. Only state of the student is changed.

The paired transitions must happen all together. If a paired transition happens independently of the other transition(s), this could leave the system in an inconsistent state. For example, consider transition number 34 in Table 5.2 (G,L,O,X,X,X → G,K,N,X,X,X) both transitions: *cancel* in student and *cancel* in section must happen together. For instance, if only *cancel* in section happen individually, the system will transition from (G,L,O,X,X,X) to (G,L,N,X,X,X) which is an inconsistent super state. The student is enrolled in a cancelled section.

The constrained transitions are performed only when one or more objects are in a specific state. They consider the super state when performing the transition. For example, in transition number 29 in Table 5.2, when perform transition *approve* the object student must be in *GoodStanding* state.

5.5 Inconsistency Discussion for UM

5.5.1 Super State Inconsistencies

We know from the domain knowledge that the student graduated when he/she finishes all courses. This is stated in invariant number 8 in section 5.3. The student who graduated cannot be in progress. The super state (R,P,X,X,X,X) is an invalid super state and should not happen.

If the student is dismissed s/he should not be eligible for enrolling in a section until the student comes back to good standing state. This condition is known from the domain knowledge and stated in invariant number 2 in section 5.3. The super state (D,E,X,X,X,X) is an invalid super state and should not happen.

The domain knowledge tells us that when a section is canceled, the enrollment is cancelled too. This is stated in invariant number 9 in section 5.3. The student cannot enroll in a canceled section. The super state (G,K,N,X,X,X) is generated and is a valid super state. On the other hand, the super state (G,L,N,X,X,X) is an invalid and will not be generated.

5.5.2 Single Step Transitions Inconsistencies

In enrollment state diagram a student cannot make the transition *approve* from state *AdvisorApproved* to state *Eligible* when the student is *OnHold*. If a single step transition such as (H,A,X,X,X,X) → (H,E,X,X,X,X) happens, it would be an invalid transition. This is because the transition set does not include such transition. In general, when the student is *OnHold* s/he is not allowed to do any enrollment activity until the *OnHold* is released by doing a payment (transition *pay)*. This is because the enrollment states are constrained by the student being on *GoodStanding* state.

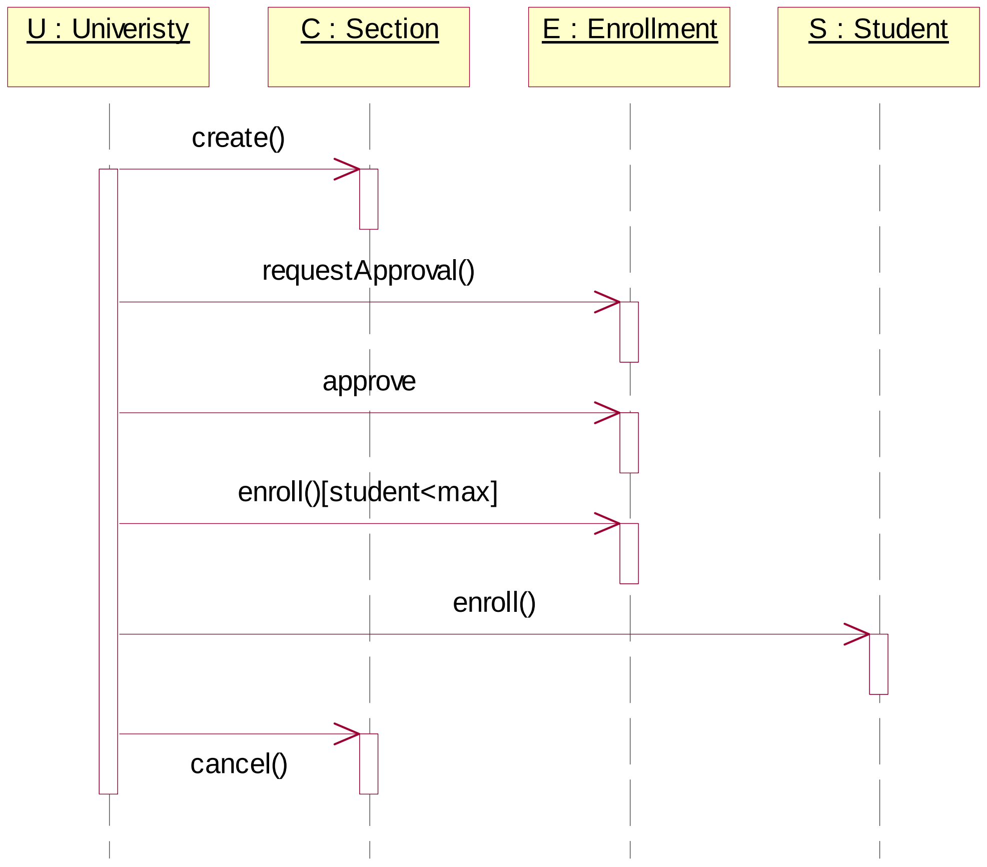
Another example for the invalid single step transition is when a section is canceled but the enrollment is not. i.e. the single step transition (G,L,O,X,X,X) → (G,L,N,X,X,X) is invalid. That is because the transition set forces the *cancel* transition in enrollment to happen simultaneously with *cancel* transition in section. The single step transition (G,L,O,X,X,X) → (G,K,N,X,X,X) is the correction for the above invalid transition.

5.5.3 Sequence Inconsistencies

The sequence diagram on Figure 5.8 shows that the university opens a new section and a student enrolls successfully in this section. The section is then cancelled but the student is still enrolled in that section. This is inconsistency because this section should be cancelled from the student schedule too. Hence the sequence:

(G,C,O,A,B,A) → (G,A,O,A,B,A) → (G,E,O,A,B,A) → (G,L,O,A,B,A) → (G,L,O,A,B,A) → (G,L,N,A,B,A) is an invalid sequence.

Figure 5.8 A Sequence Diagram for a Class Enrollment



CHAPTER 6 - SCALABILITY

Since *Super State Analysis* uses all possible combination of states to generate the sequences, this could cause a state explosion during the generation of the sequences. The state explosion problem is a well-known problem in the area of computation [VA98]. Many researchers have attempted to find techniques to reduce the state explosion in different areas [GA05, RA06, HO07, ST01].

In *Super State Analysis,* often more than one instance of each class is involved in the analysis to discover the inconsistencies. Using only one instance of each class in the analysis may miss some inconsistencies that may arise when using more than one instance of some classes. For example, assume that there is a system with two classes: C_1 and C_2 interacting together and having some paired transitions. Assume that class C_1 can interact with *n* instances of class C_2. It is better to have *n* instances of class C_2 to detect the inconsistency that will not be detected when using only one instance of each. However, using *n* instances of each class will increase the number of super states. In general, the total number of super states involved in *Super State Analysis* is calculated using the following equation:

Total number of states = $S_1^{C1} * S_2^{C2} * \ldots * S_n^{Cn}$

Where:

C_i is the number of instances of class *i* that involved in *SSA*

S_i is the number of states in class C_i

n is the number of classes

There are several techniques that could be applied to *Super State Analysis* to reduce the state explosion. The paired transitions technique is used to select a smaller number of instances of some objects. It is not always necessary to analyze *n* instances of each object. Instead, by studying the behavior and interaction between the classes, a smaller number may be used. This may make a large reduction in the total number of states since we decrease the C_i in S_i^{Ci}.

There are some other possible techniques that will be discussed that can be applied to *super state analysis*. One possible technique involves reducing the number of objects in the system by eliminating unnecessary objects from the analysis. Another possible technique is decreasing the number of states in some classes. Each class can be analyzed and some of its states may be merged together to reduce the total number of states. The final possible technique discussed is limiting the number of steps in each sequence to reduce the number of sequences.

6.1 Paired Transitions Technique

The paired transitions are the transitions that must happen together. The paired transitions were discussed in chapter 3. For example, in the library example in chapter four, transition *checkOut* in Patron is paired with transition *checkOut* in Book. It is similar for transition *return* in Patron and Book. The paired transitions can be used as a guide to select the number of instances of each class. The classes that are involved in paired transitions may have more than one instance in the analysis.

The total number of states can be reduced by analyzing a smaller number of instances of some classes. Selection of a smaller number of instances will reduce the state explosion. The selection of a smaller number of instances of each class needs some analysis for the system to decide the selection. The interaction between the classes should be considered to select the number of instances. For example, consider a system with two classes: C_1 and C_2 with a restriction that C_1 may interact with at most *n* instances of class C_2. Using paired transition technique, we can chose one instance of C_1 and *n* instances of C_2 instead of analyzing *n* instances of each. This selection will reduce the number of super state and therefore reduce the state explosion.

In the library example in chapter 4, there are two objects: Patron which has three states and Book which has six states. Assume that the patron can checkout at most two books. When using *super state analysis* with two instances of book and two instances of patron, *SSA* will generate 324 states. However, doing first, two instances of book and one instance of patron will generate 108 states and then doing two instances of patron and one instance of book will generate 54 states. Doing one instance of patron and two instances of book or one instance of book and two instances of patron may greatly reduce the state explosion. Table 6.1 shows the effect of object selection on the total number of super states for the library example. In general, reducing the number of instances by one instance will reduce the state explosion by S_i where S_i is the number of states in class *i*.

Table 6.1 Effect of object selection on the total number of super states

Number of Objects	Total Number of Super States
Patron, Book1, Book2	108
Patron1, Patron2, Book	54
Patron1, Patron2, Book1, Book2	324

6.2 Object Reduction Technique

This technique can be applied to *Super State Analysis* to reduce the number of objects. If there are some objects acting independently, they can be eliminated from the analysis. An object is independent when it does not affect the state of the other object and its state is not affected by other objects. The independent objects can be identified from the transition set. If the state of an object has always ‘X’ in pre state and post state, this object is independent from other objects and it can be eliminated from the analysis to reduce the state explosion. The transition set on Table 6.2 shows the behavior of the independent object. Object ‘O_2’ is an independent object since its state is always ‘X’ regardless of the states of the other objects.

Table 6.2 Behavior of independent objects in transition set

SS_{pre} $O_1,O_2,O_{3,}\ldots,O_n$	$\rightarrow$	SS_{post} $O_1,O_2,O_{3,}\ldots,O_n$	Transition(s)
$S_1,X,S_1,\ldots,S_1$	$\rightarrow$	$S_2,X,S_2,\ldots,S_1$	t_1,t_3
$S_2,X,S_2,\ldots,S_1$	$\rightarrow$	$S_2,X,S_3,\ldots,S_1$	t_4
$S_2,X,S_3,\ldots,S_1$	$\rightarrow$	$S_3,X,S_3,\ldots,S_2$	$t_{5,}$ t_6
$\vdots$	$\rightarrow$	$\vdots$	$\vdots$

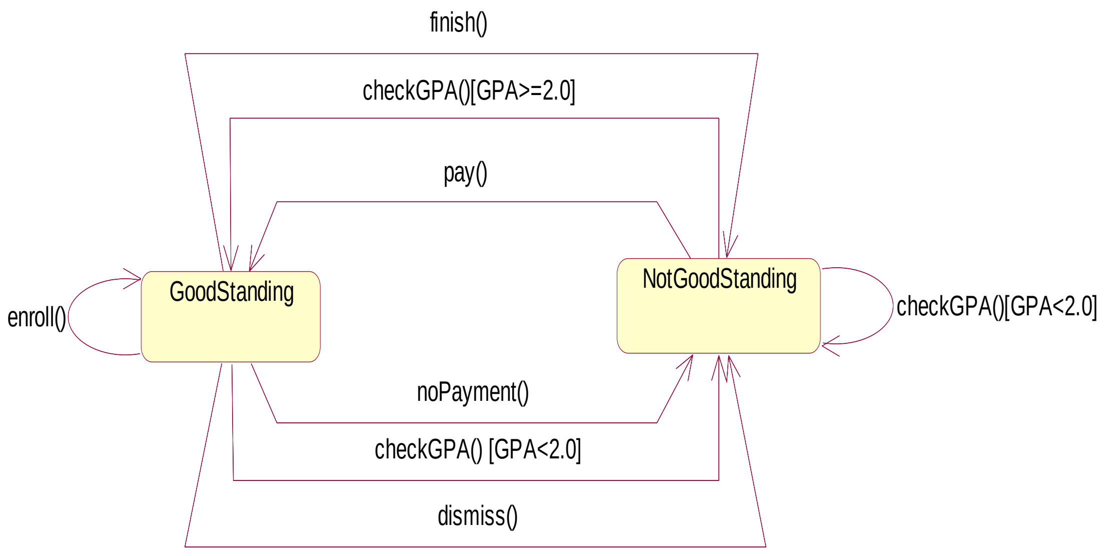
For example, assume that for the library example in chapter 4 there is another state diagram for *staff*. So, we have a total of three state diagrams: *Patron, Book,* and *Staff*. If object *Staff* does not have any interaction with neither *Patron* nor *Book*, it will always have ‘X’ when patron or book makes a transition. Thus, object *Staff* can be eliminated from the analysis.

6.3 State Reduction Technique

The number of states in each state diagram involved in *Super State Analysis* could be decreased by merging some states together. The state reduction is an applicable technique and can be applied to *Super State Analysis* to reduce the state explosion. Reduction will require the developer to analyze the transition set and decide which group of states can be merged together.

The university model on chapter five has 6 state diagrams (Figure 5.2 - Figure 5.7). Some of these diagrams may have some states that can be merged together to reduce the number of states. For Example, the student state diagram on Figure 5.4 can be reduced to two states: *GoodStanding* and *NotGoodStanding.* This will reduce the number of state from six states to only two states. The reduced state diagram for Student is shown on Figure 6.1.

Figure 6.1 Reduced State Diagram for Student



6.4 Limit the number of steps Technique

Another applicable way to reduce the state explosion is to limit the number of steps in each sequence. We may limit the sequence computation to a smaller number to reduce the state explosion. For example, the *Super State Analysis* could be restricted to perform the analysis up to a certain number of steps instead of doing all possible steps. However, limiting the number of steps will not guarantee to discover all possible inconsistencies.

CHAPTER 7 - SPECIFICATION AND IMPLEMENTATION

7.1 Specification

In this chapter, set notation is used to specify *Super State Analysis*. I use set notation to compare the different sets formally. I identify the relationships between these different sets. *Super State Analysis* contains five comparisons between eight different sets. Specifically, using set notations, I compare the following sets of *SSA* model in Figure 3.1 in page 32.

- set T1 with set H1

  Set of all generated Super States **vs.** *Set of valid Super States (SS*)

- set T1 with set H2

  Set of all generated Super States **vs.** *Set of invalid Super States (SS*)

- set T2 with set H3

  Set of all generated single step transitions **vs.** *Set of valid single step transitions*

- set T2 with set H4

  Set of all generated single step transitions **vs.** *Set of invalid single step transitions*

- set T3 with set S

  Set of all generated sequences **vs**. *Sequences*

7.2 Formalization of *Super State Analysis (SSA)*

A system is specified by class diagrams, sequence diagrams, and state diagrams. In the real world, a system may have more diagrams but we are here interested in these three diagrams.

A system $\mathbb{S} = \{Cl, Seq, St\}$ where
Cl is a set of class diagrams,
Seq is a set of sequence diagrams, and
St is a set of state diagrams.

A class diagram describes the static structure of the system. A class diagram, *Cdig*, is a set of classes. The associations between classes are not of concern for this analysis. There are three components for each class: *name*, set of *methods*, and set of *attributes*.

A class diagram, $Cdig \in Cl$, is a set of classes, Cls.
Each class, $C \in Cls$, has three elements:
$C = \{cname, Mthd_{cname}, Att_{cname}\}$
where
$cname$: the class name,
$Mthd_{cname}$: the set of all methods of class C, and
Att_{cname}: the set of all attributes of class C

A sequence diagram is a sequence of *calls* between classes that occur in a time sequence. There are three components for each *call* in a sequence diagram:

1. The source class of the call.

   The source class must be a class in the class diagram.

2. The destination class of the call.

   The destination class must be a class in the class diagram.

3. The message.

   To insure consistency with the class diagram, the message must be a method in the destination class.

A sequence diagram, *Sdig* $\in$ *Seq*, is an ordered tuple of calls. Each *call* $\in$ *Sdig*, has the form:
call = [call$_{src}$, call$_{msg}$, call$_{des}$]
where
call$_{src}$: the source class such that call$_{src}$ $\in$ *Cls*
call$_{msg}$: the message call such that call$_{msg}$ $\in$ $Mthd_{\text{call}_{des}}$
call$_{des}$: the destination class such that call$_{des}$ $\in$ *Cls*

Let R be the set of all sequences that appear in a sequence diagram of system $\mathbb{S}$.
A sequence $r \in$ R contains tuples of *call.*
$\forall\ r \in$ R, $r = <call_0, call_1, \ldots, call_{n-1}>$ where n is the number of calls that appear in sequence *r.*

The state diagram describes the different states of an abject. It contains transitions and states. The state of an object may change by a transition. Each transition on the state diagram has a matching method on class diagram. A state diagram has initial and terminal states. From every initial state we can get to every state and from every state we can get to a terminal state.

A state diagram, *StDig* $\in$ *St*, is a set that has five elements:
{S$_{all}$, S$_{initial}$, S$_{terminal}$, transitions, T}
Where
S$_{all}$: set of all states in the state diagram *StDig*
S$_{initial}$: set of all initial states such that S$_{initial}$ $\subseteq$ S$_{all}$
S$_{terminal:}$ set of all terminal states such that S$_{terminal}$ $\subseteq$ S$_{all}$
T is a tuple that has the form: [S_{pre}, *trans,* S_{post}]
where
S_{pre}: the state before transition *trans* such that S_{pre} $\in$ S$_{all}$

trans: the transition such that *trans* $\in$ transitions
S_{post}: the state after transition *trans* executed such that $S_{post} \in S_{all}$
$\forall s_a \in S_{all}$ *and* $s_l \in S_{initial}$ $\exists$ tuples $\in$ T such that $[s_l, \{t\}, s_a]$ *and* $\{t\} \subseteq T$
$\forall s_a \in S_{all}$ *and* $s_t \in S_{terminal}$ $\exists$ tuples $\in$ T such that $[s_a, \{t\}, s_t]$ *and* $\{t\} \subseteq T$

The Super State of a system combines the state information of multiple state diagrams into a composite state. The super state describes the state of the whole system. The super state may be changed after a transition or after paired transitions. A system may have many different super states depending on the selection of classes that are being analyzed. It is not necessary to analyze all the classes. We have discussed some selection techniques in details on chapter 6. Also, the selection of how many instances of each class affect the super state since *SS* contains all individual states of each instance involved in the analyzed system. Choosing different number of classes and instances results in different number of super states.

Let *SS* be a super state of system $\mathbb{S}$. *i.e.* the partial state of a whole system. For system $\mathbb{S}$, the super state *SS* has the form $[s_1, s_2, \ldots, s_n]$ where s_i is the state of object *i* and *n* is the number of objects in the system $\mathbb{S}$.
SS may be changed by transitions, so we have $<SS_{pre}, t, SS_{post}>$ where
SS_{pre} : is the super state before transition *t*
t : is a transition
SS_{post}: is the super state after transition *t* executed

Sequence diagrams do not show states of object. However, there are implicit *Super States* between the calls. We may add *SS* between calls by getting the appropriate states for *SS* from state diagrams. Some of the states in *SS* may not be completely specified. For example, when the state of some object is ‘x’. After each *call* we look for the state diagram of the destination class and get all possible states for this class after this *call*. After the call, the destination class may be changed to one of its states. The states of other objects do not changed.

> For every *call* in the sequence diagram we have $<SS_{pre}, call, SS_{post}>$ where SS_{pre} is the super state before *call* and SS_{post} is the super state after the message *call* has been called.
>
> In SS_{post}, only state of at most one class is changed. This class must be the destination class of the message call, $call_{des}$. The state of other objects remains in the same state as they were before *call*.
>
> Let R’ be the set of all sequences which have *Super States* included.
> For each $r \in$ R, we have one or more matching $r' \in$ R’ where the message sequence of calls in *r* is the same message sequence in *r’*.
> Now we have sequence $r' \in$ R’ which has *SS* included.

The sequence starts with a super state. The super state may be changed after each call in the sequence. In each super state in the sequence, only the state of the destination object of the call may change.

$\forall\ r' \in R'$, $r' = <SS_0, call_0, SS_1, call_1, \ldots, SS_{n-1}, call_{n-1}, SS_n>$ where n is the number of calls that appear in sequence r and SS_i is the super state after transition t_{i-1} and $0 \leq i \leq n$.

The super state is changed from state to another by legal transitions. We use the transition matrix technique to generate all sequences of legal transitions. By computing the transition matrix closure of the legal transitions of system $\boldsymbol{S}$ we generate all possible sequences.

Let G be the set of generated sequences. A sequence $g \in G$ contains transitions separated by super state SS. In this sequence, the super state SS may change after each transition. The sequence $g \in G$ starts from any valid state and ends with any reachable state.

$\forall\ g \in G$, $g = < SS_0, t_0, SS_1, t_1, \ldots, SS_{m-1}, t_{m-1}, SS_m>$ where m is the number of transitions that appear in sequence g and SS_i is the state after transition t_{i-1} and $0 \leq i \leq m$

Assume that for each message *call* on sequence diagram, there is at least one matching transition *t* on state diagram. Furthermore, each transition *t* on state diagram has a matching method on class diagram and each message *call* on sequence diagram has a matching method on class diagram.

Claim 1:

Every valid sequence *r* $\in$ R has a matching subsequence *g* $\in$ G.

Any sequence in a valid sequence diagram should be a subsequence of the set of sequences that are possible in the generated sequences. The valid sequences that appear in a sequence diagram will have matching sequences in the generated sequences. This is because all possible sequences are generated in the set of generated sequences. Hence, any valid sequence must have at least one matching subsequence in the generated sequences.

Consider an arbitrary sequence $r \in$ R. For r, there is a matching sequence $r' \in$ R'. The sequence r' has the form $<SS_0, call_0, SS_1, call_1, \ldots, SS_{n-1}, call_{n-1}, SS_n>$.

Let SS_0 be the initial state for sequence r'. SS_0 is not necessary an initial state for system $\mathcal{S}$. Then there exist a first call in the sequence, $call_0$, which changes the state to SS_1. Because $call_0$ is a legal message call, the super state will change from SS_0 to SS_1.

Because G contains all the generated sequences, G will have at least one subsequence $g \in$ G which starts with $<SS_0, t_0, SS_1>$ where $t_0 = call_0$ and the state of destination class in sequence diagram is changed in SS_1. Otherwise, the sequence r' is an invalid sequence. *i.e.* If there is no such subsequence in G.

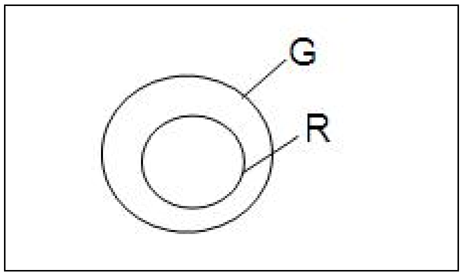
Since the state diagrams may include guarded transitions and transitions that don't change the states (a.k.a. No-Op transitions), the generated sequences in G may have tuples of the forms $<SS_{pre}, t, SS_{post1}>$ and $<SS_{pre}, t, SS_{post2}>$ and $<SS_{pre}, t, SS_{pre}>$. That is because all the possibility of the guarded transitions will be generated as well as the transitions that do not change the super state.

Assume that $<SS_i, call_i, SS_{i+1}>$ in *r'* has a matching tuple in *g*. For each tuple $< SS_{i+1}, call_{i+1}, SS_{i+2} >$ in *r'*, if $call_{i+1}$ is a valid message *call* that changes the super state from SS_{i+1} to SS_{i+2}, we will definitely have a matching tuple in *g* with $t_{i+1} = call_{i+1}$ and state of destination class in sequence diagram is changed in SS_{i+2}. The sequence *r'* would be an invalid sequence if it does not match any subsequence in G. This is because G generates all possible sequences. In case of guarded transitions and No-Op transitions, sequence *r'* will include one possibility of sequences instead of having all possible sequences. Hence, for every valid tuple in *r'* we will have one or more matching tuples in *g*.

Assume that in step *n* of the sequence *r'* there is a matching tuple in *g*. So, we have the tuple $<SS_n, call_n, SS_{n+1}> \in$ g. Since *r'* is a valid sequence and *g* is the matching generated sequence, the tuple $<SS_{n+1}, call_{n+1}, SS_{n+2}> \in$ g. Therefore, by induction the sequence in *r'* has matching sequence in *g*.

So, valid sequences in R will have matching subsequences in G. This is because G contains all generated sequences. However, there are some generated sequences that do not appear in R. That is because R has only the sequences that appear in the sequence diagrams of $\mathcal{S}$. If a designer draws an incorrect sequence diagram, this sequence is an invalid sequence. The invalid sequences in R will not have matching sequences in G because G will include only the valid sequences. Hence, we can write that R $\subseteq$ G. The relationship between R and G is shown in Figure 7.1.

Figure 7.1 The relationship between R and G



Let T1 be the set of all generated super states,

Let H1 be the set of valid super state, and

Let H2 be the set of invalid super state.

Claim 2:

Each valid *super state* is included in the set of all generated super states

A valid *super state* should be in the set of all generated super states. This is because all possible super states are generated in the set of generated SS.

Assume that $h \in$ H1 is a valid super state but $h \notin$ T1. Set T1 is the set of all generated super states. Thus, h must be generated in T1. This contradicts our assumption. Hence, $h \in$ T1.

Claim 3:

Each invalid super state is excluded from the set of all generated super states

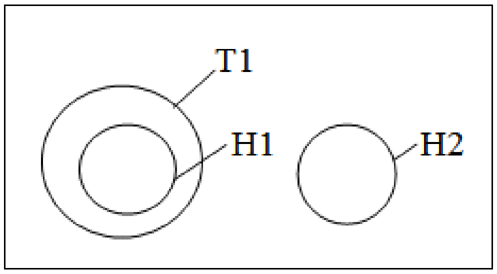
An invalid *super state* should not be in the set of all generated super states. This is because only the valid super states are generated in the set of generated SS.

Assume that $h \in$ H2 is an invalid super state and $h \in$ T1. Set T1 is the set of all generated super states. Thus, h must not be generated in T1 because it is an invalid super state. This contradicts our assumption. Hence, $h \notin$ T1.

From Claim 2 above we can observe that H1 $\subset$ T1. Similarly, from Claim 3 we can observe that T1 and H2 are disjoint sets. The relationship between T1, H1, and H2 is shown in Figure 7.2 . As a result we can write:

- T1 $\cap$ H1 = H1
- T1 $\cap$ H2 = ϕ

Figure 7.2 The relationship between T1, H1, and H2



Let T2 be the set of generated single step transitions.

Let H3 be the set of valid single step transitions.

Let H4 be the set of invalid single step transitions.

Claim 4:

Each valid single step transition is included in the set of all generated single step transitions

A valid single step transition should be in the set of all generated single step transitions. This is because all possible single step transitions are generated in the set of generated single step transitions.

Assume that $h \in$ H3 is a valid single step transition but $h \notin$ T2. Set T2 is the set of all generated single step transitions. Thus, h must be generated in T2. This contradicts our assumption. Hence, $h \in$ T2.

Claim 5:

Each invalid single step transition is excluded from the set of all generated single step transitions

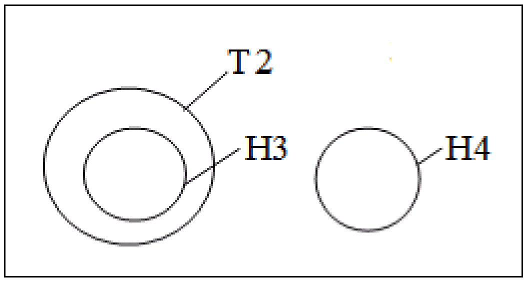
An invalid single step transition should not be in the set of all generated single step transitions. This is because only the valid single step transitions are generated in the set of generated SS.

Assume that $h \in$ H4 is an invalid single step transition and $h \in$ T2. Set T2 is the set of all generated single step transitions. Thus, h must not be generated in T2 because it is an invalid single step transition. This contradicts our assumption. Hence, $h \notin$ T2.

From Claim 3 above we can observe that H3 $\subset$ T2. Similarly, from Claim 4 we can observe that T2 and H4 are disjoint sets. The relationship between T2, H3, and H4 is shown in Figure 7.3. As a result we can write:

- T2 $\cap$ H3 = H3
- T2 $\cap$ H4 = ϕ

Figure 7.3 The relationship between T2, H3, and H4



7.3 Implementation

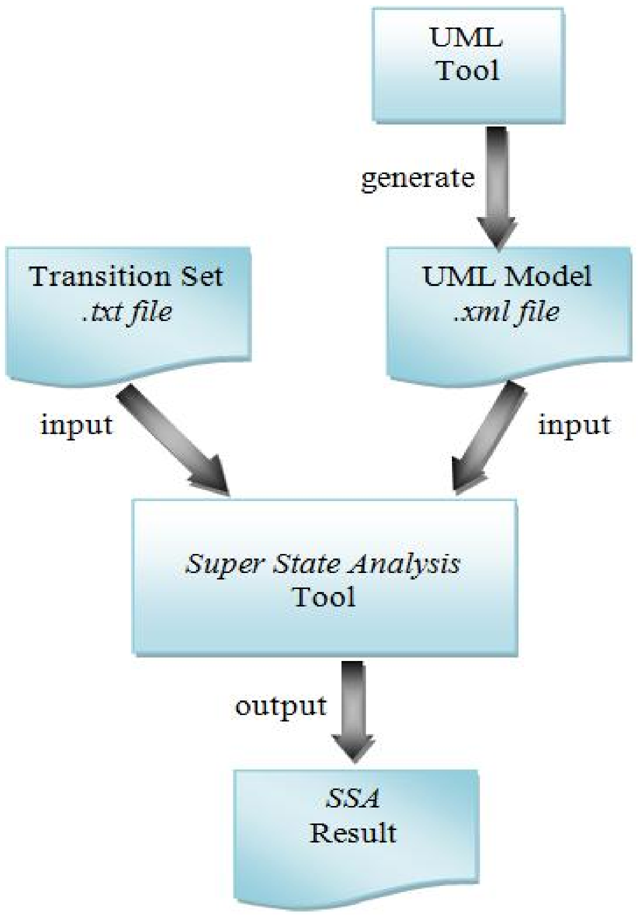
7.3.1 Tool Description

The *Super State Analysis* Tool checks consistency between multiple UML state diagrams and sequence diagram. The tool is supplied with two files as a tool input. The first file contains an XML representation of the UML state and sequence diagrams. The XML file can be generated from a UML tool. The second file contains a transition set that is user defined. The transition set is a text based file created by the user. The tool performs the analysis and detects the sequence inconsistencies if there is any. The output is displayed in text upon the completion. The tool architecture is shown in Figure 7.4. The transition set file has the following format:

$$[\ S_{\mathrm{pre}1}, \ldots, S_{\mathrm{pre}n} > \text{Transition(s)} > S_{\mathrm{post}1}, \ldots, S_{\mathrm{post}n}\ ]$$

If a user has no preference of the state of a particular object in a super state, then an ‘x’ can be used to denote “don’t care.”

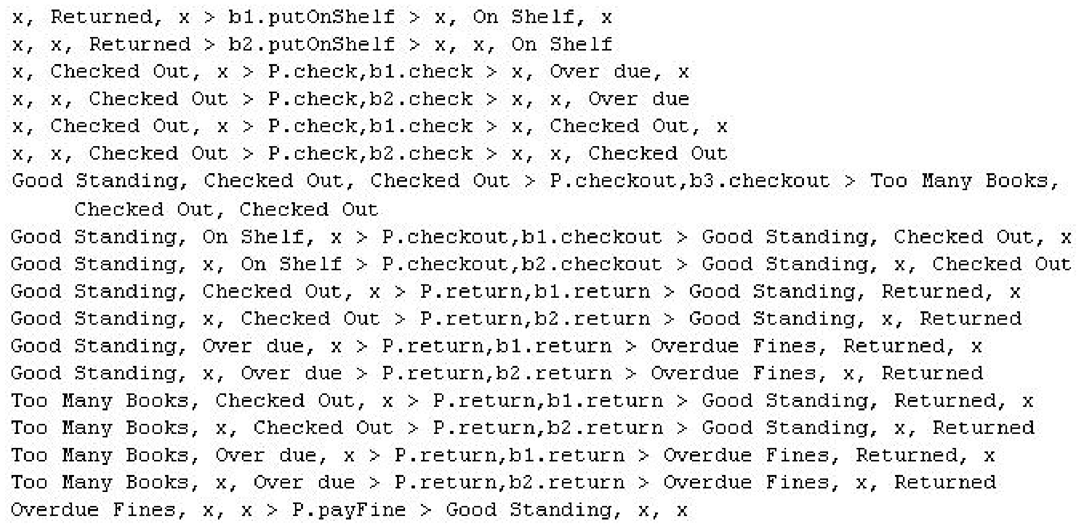
Figure 7.4 *SSA* Tool Architecture



7.3.2 Tool Example

This example is based on the library example described earlier in chapter 4. We supplied the tool with the two state diagrams shown in chapter 4 (Figure 4.2 and Figure 4.3). Also, we supplied the tool with the transition set that shown in Figure 7.5.

Figure 7.5 Transition Set File



```
x, Returned, x > b1.putOnShelf > x, On Shelf, x
x, x, Returned > b2.putOnShelf > x, x, On Shelf
x, Checked Out, x > P.check,b1.check > x, Over due, x
x, x, Checked Out > P.check,b2.check > x, x, Over due
x, Checked Out, x > P.check,b1.check > x, Checked Out, x
x, x, Checked Out > P.check,b2.check > x, x, Checked Out
Good Standing, Checked Out, Checked Out > P.checkout,b3.checkout > Too Many Books,
     Checked Out, Checked Out
Good Standing, On Shelf, x > P.checkout,b1.checkout > Good Standing, Checked Out, x
Good Standing, x, On Shelf > P.checkout,b2.checkout > Good Standing, x, Checked Out
Good Standing, Checked Out, x > P.return,b1.return > Good Standing, Returned, x
Good Standing, x, Checked Out > P.return,b2.return > Good Standing, x, Returned
Good Standing, Over due, x > P.return,b1.return > Overdue Fines, Returned, x
Good Standing, x, Over due > P.return,b2.return > Overdue Fines, x, Returned
Too Many Books, Checked Out, x > P.return,b1.return > Good Standing, Returned, x
Too Many Books, x, Checked Out > P.return,b2.return > Good Standing, x, Returned
Too Many Books, Over due, x > P.return,b1.return > Overdue Fines, Returned, x
Too Many Books, x, Over due > P.return,b2.return > Overdue Fines, x, Returned
Overdue Fines, x, x > P.payFine > Good Standing, x, x
```

We compare the state diagrams with two sequences diagrams. The first sequence diagram in Figure 7.6 is checking out two books and returning an overdue book. The tool's output in Figure 7.7 shows that the sequence in Figure 7.6 was not legal since a check action on the book must occur before a book becomes overdue.

Figure 7.6 Sequence for returning overdue book

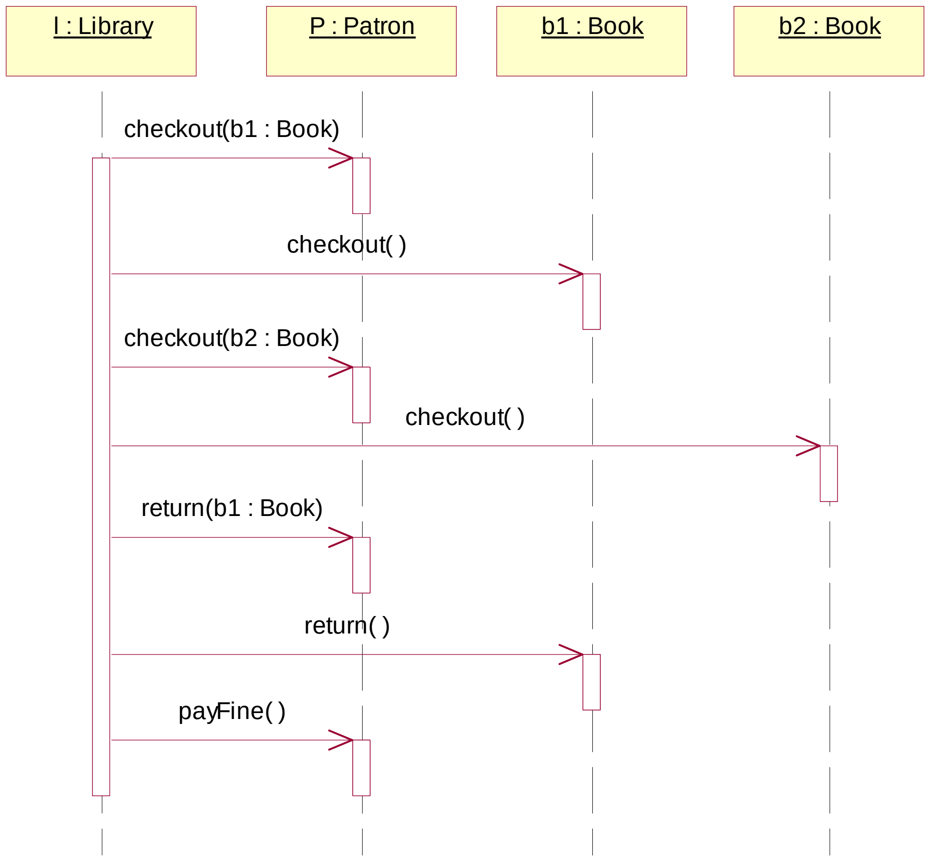


Figure 7.7 Tool output for Figure 7.6 sequence diagram

```
The Sequence Model '({Logical View}test1)' to State Models (Patron, Book, Book)
comparison does not contain the list of transitions: 'P.checkout, b1.checkout,
P.checkout, b2.checkout, P.return, b1.return, P.payFine'.
```

The second sequence diagram in Figure 7.8 is checking out two books and returning an overdue book. Figure 7.9 shows that the tool correctly identified the sequences in Figure 7.8 as a correct set of sequences.

Figure 7.8 A corrected sequence for overdue book

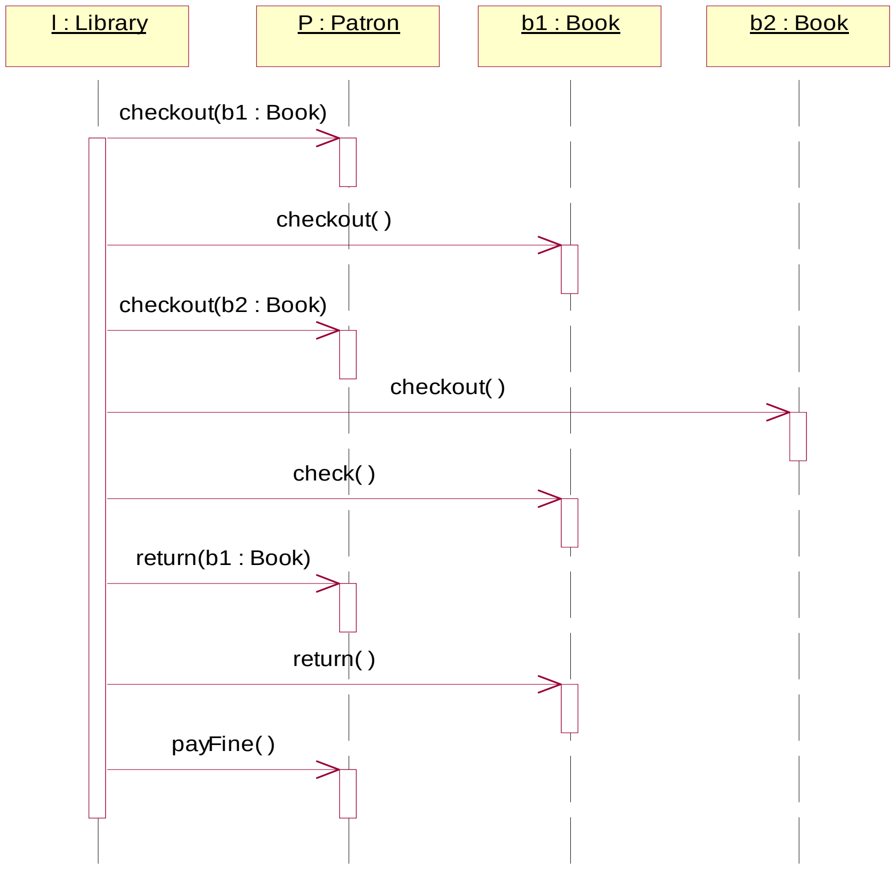


Figure 7.9 Tool output for Figure 7.8 sequence diagram

```
The Sequence Model '({Logical View}test2)' to State Models (Patron, Book, Book)
comparison found no errors.
```

I have tested the tool with the library example (chapter 4) in three different cases. The first case is when we have one book and two patrons. The total number of super states in this case is 54 states. Figure 7.10 shows the execution time for this case with different number of closure. In the second case, we pick two books and one patron. The total number of super states in this case is 108 states. Figure 7.11 shows the execution time for this case with different number of closure. In the third case, we tested the system with two books and two patrons. The total number of super states in this case is 324 states. Figure 7.12 shows the execution time for this case with different number of closure. The tool was tested under Microsoft Windows XP Professional with a machine that has Intel Pentium 3.00 GHz and 2 GB of RAM.

Figure 7.10 Execution time for 1 Book and 2 Patrons

Number of closure	Execution time (seconds)
1	0.078
2	0.109
3	0.141
4	0.157
5	0.188
6	0.391

Figure 7.11 Execution time for 2 Books and 1 Patron

Number of closure	Execution time (seconds)
1	0.125
2	0.156
3	0.234
4	0.422
5	2.360
6	3.922

Figure 7.12 Execution time for 2 Books and 2 Patrons

Number of closure	Execution time (seconds)
1	0.251
2	1.57
3	2.141
4	3.990
5	10.331
6	50.399

CHAPTER 8 - CONCLUSION

Unified Modeling Language has been used as a standard language for software modeling. It consists of 13 types of diagrams. Each diagram is used for a different design aspect. Usually many diagrams are involved in software development. Using more than one diagram to design a system is necessary but can leave the system in an inconsistent state and hence produce errors. Finding inconsistencies in software design before the design is implemented is very important. We should check the consistency among the diagrams and make sure that the diagrams are consistent.

The information in UML diagrams are related to each other and represent different views of a system. Hence, they can be validated against each other. n this dissertation, I have proposed a new approach to check the consistency between multiple state diagrams and one or more sequence diagrams using *Super State Analysis* (SSA). This super state details all of the possible composite states the objects can be in as well as the transition pairs which lead from one composite state to another. The analysis discovers inconsistencies that cannot be detected when considering only a single state diagram. *Super State Analysis* identifies the five types of inconsistencies that are related to state and sequence diagrams:

- Valid super states
- Invalid super states
- Valid single step transitions
- Invalid single step transitions

- Invalid sequences

Super State Analysis model uses a transition set that captures relationship information that is not specifiable in UML diagrams. The *SSA* model uses the transition set to link transitions of multiple state diagrams together. The analysis generates three different sets automatically. These generated sets are compared to the provided sets to detect the inconsistencies. Comparing the information from different sources allows us to detect the inconsistencies. *Super State Analysis* performs five types of comparisons to detect the inconsistencies.

There are several techniques could be applied to *Super State Analysis* to reduce the state explosion. The paired transitions technique is used to select a smaller number of instances of some objects. It is not always necessary to analyze *n* instances of each object. Instead, by studying the behavior and interaction between the objects, a smaller number may be used. There are some other possible techniques that can be applied to *super state analysis*. Some possible technique involves reducing the number of objects in the system, decreasing the number of states in some objects, and limiting the number of steps in each sequence to reduce the number of sequences.

In the future, the *Super State Analysis* can be fully automated. The comparisons C1, C2, C3, and C4 in *Super State Analysis* model (Figure 3.1) can be fully automated if we formalize the four sets: H1, H2, H3, and H4 and feed them to the system. By comparing these four sets to the generated sets: T1 and T2 the super state inconsistencies and single step transitions inconsistencies can be detected automatically. Moreover, the

Super State Analysis tool can be integrated with some UML tool (e.g. Rational® Software Architect) to perform the consistency checking directly and instantly within the UML tool.

REFERENCES

[AL06] Mohammad Alanazi, Jason Belt, and David Gustafson, *“UML Analysis Using State Diagrams”*, Proceedings of the International Conference on Software Engineering Research and Practice, Vol. 2, pp. 569-576, Las Vegas, NV, June 2006.

[AL07] Mohammad Alanazi and David Gustafson, *“Comparing Multiple State Diagrams to Sequence Diagrams using Super State Analysis”*, Proceedings of the 11th International Conference on Software Engineering and Applications, pp. 494-500, Cambridge, MA, November 2007.

[AL08a] Mohammad Alanazi and David Gustafson, *“Inconsistency Discovery in Multiple State Diagrams”,* International Journal of Computer Science and Engineering, pp. 153-161, Vol. 2, Number. 3, May 2008.

[AL08b] Mohammad Alanazi and David Gustafson, *“Error Detection in Multiple State Diagrams”*, Proceedings of the International Conference on Software Engineering Research and Practice, Vol. 1, pp 184-190, Las Vegas, NV, July 2008.

[AL08c] Mohammad Alanazi and David Gustafson, *“Inconsistency Discovery in Multiple State Diagrams”,* Proceedings of World Academy of Science and Technology, pp. 54-62, Vol. 28, 2008.

[AM04] Scott W. Ambler, “*The Object Primer: Agile Model-Driven Development with UML 2.0*”, 3rd Edition, Cambridge University Press, 2004.

[AR08] IBM® , *Rational® Software Architect*, http://www-01.ibm.com/software/awdtools/architect/swarchitect/index.html

[BO05] Bontemps, Y.; Heymans, P.; Schobbens, P.-Y, *"From Live Sequence Charts to State Machines and Back: A Guided Tour"*, IEEE Transactions on Software Engineering, 31(12): pp. 999--1014, December 2005.

[DU00] Yves Dumond, Didier Girardet , Flavio Oquendo, *"A relationship between sequence and statechart diagrams"*, Dynamic Behaviour in UML Models: Semantic Questions, UML 2000 Workshop.

[EG01] Alexander Egyed, *"Scalable Consistency Checking between Diagrams – The ViewIntegra Approach"*, Proceedings of the 16th Annual International Conference on Automated Software Engineering, pages 387--390, 2001.

[EG06] Alexander Egyed, "*Instant consistency checking for the UML"*, Proceeding of the 28th international Conference on Software Engineering, Pages 381-390, 2006.

[ER03] Hans-Erik Eriksson, Magnus Penker, Brian Lyons, and David Fado, "*UML 2 Toolkit*", John Wiley and Sons, Inc., New York, NY, October 2003.

[GA05] Qitao Gan, Bjarne E. Helvik, "*Limiting the State Space Explosion as Taking Dynamic Issues into Account in Network Modeling and Analysis*", *Norwegian Network Research Seminar (NNRS2005)*, Fornebu, Norway, October 27-28, 2005.

[GO03] Hassan Gomaa and Duminda Wijesekera, "*Consistency in Multiple-View UML Models: A Case Study*", Proceeding of Workshop on Consistency Problems in UML-based Software Development, 6th International Conference on the Unified Modeling Language, San Francisco, October 2003.

[HO07] Viliam Holub, "*Fighting the state explosion problem in component protocols*", PhD Thesis, Department of Software Engineering, Charles University in Prague, Czech Republic, 2007.

[KI04] Soon-Kyeong Kim and David Carrington, "*A Formal Object-Oriented Approach to defining Consistency Constraints for UML Models"*, Proceedings of the 2004 Australian Software Engineering Conference, pages 87--94, 2004.

[KR00] Padmanabhan Krishnan, *"Consistency Checks for UML"*, Proceedings of the Seventh Asia-Pacific Software Engineering Conference, pages 162--169, 2000.

[KU03] Ludwik Kuzniarz and Miroslaw Staron, *"Inconsistencies in Student Designs"*, In the Proceedings of The 2nd Workshop on Consistency Problems in UML-based Software Development, pp. 9-18, San Francisco, CA, 2003.

[LA03] C. Lange, M.R.V. Chaudron, J. Muskens, L.J. Somers and H.M. Dortmans, "*An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs*", San Francisco, October 2003.

[LI03] Boris Litvak, Shmuel Tyszberowics, and AmiramYehudai, *"Behavioral Consistency Validation of UML Diagrams"*, Proceedings of the First International Conference on Software Engineering and Formal Methods, pages 118--125, 2003.

[OM06] OMG Unified Modeling Language Specification, *UML 2.0*, Object Management Group, 2006, http://www.uml.org**.**

[PA01] Zs. Pap and I. Majzik and A. Pataricza and A. Szegi, *"Completeness and Consistency Analysis of UML Statechart Specifications"*, 2001.

[PI03] Orest Pilskalns, Anneliese Andrews, Sudipto Ghosh, and Robert France, *"Rigorous Testing by Merging Structural and Behavioral UML Representations"*, UML 2003 - The Unified Modeling Language: Modeling Languages and Applications, 2863: 234--248, 2003.

[RA06] Datla Vijaya Gopala Raju, Kakarlapudi Venkata Satya Varaha Narsimha Raju, and Avula Damodaram, "*The RATG System: Reducing Time with an Approach of Testing based on Combinatorial Design Method"*, Proceedings of the 24th International Conference on Software Engineering, pp. 361-366, Innsbruck, Austria, February 2006.

[SH06] Wuwei Shen, Weng Liong Low, *"Consistency Checking Between Two Different Views Of a Software System"*, Proceedings of the 10th IASTED international conference software engineering and applications, Dallas, TX, November 2006.

[ST01] Douglas A. Stuart, Monica Brockmeyer, Aloysius K. Mok, and Farnam Jahanian, "*Simulation-Verification: Biting at the State Explosion Problem*", IEEE Transaction on Software Engineering, pp. 599-617, Vol. 27, No. 7, July 2001.

[ST04] Ragnhild Van Der Straeten, Jocelyn Simmonds and Viviane Jonckers, *"Maintaining Consistency between UML Models Using Description Logic"*, Journal S'erie L'objet - logiciel, base de donn'ees, r'eseaux, Pages 231-244. 2004.

[VA98] Antti Valmari, "*The State Explosion Problem"*, Lectures on Petri Nets I: Basic Models, Lecture Notes in Computer Science, Vol. 1491, Springer-

Verlag 1998, pp. 429-528.

[WA03] Robert Wagner, Holger Giese, and Ulrich Nickel, “*A Plug-In for Flexible and Incremental Consistency Management*”, in Proc. of the International Conference on the Unified Modeling Language 2003 (Workshop 7: Consistency Problems in UML-based Software Development), San Francisco, October 2003.

[WA05] Hongyuan Wang, Tie Feng, Jiachen Zhang, and Ke Zhang, “*Consistency check between behaviour models”*, Proceedings of the International Symposium on Communications and Information Technology, pages 486 - 489, 2005.

[WI08] Wikipedia: the free encyclopedia , http://en.wikipedia.org, 2008.