uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Xu Cheng

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M. (Computer Science)

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Supporting Automated System-level Test Scenario Generation

TITRE DE LA THÈSE / TITLE OF THESIS

Stephane Somé

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Alan Williams

Samuel Ajila

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

Supporting Automated System-level Test Scenario Generation

Xu Cheng

B.C.S, Shanghai University, China
B.B.A, Shanghai University, China

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies (F.G.P.S)
in partial fulfillment of the requirements for the degree of

Master of Computer Science (M.C.S.)

under the auspices of the
Ottawa-Carleton Institute for Computer Science (OCICS)

March 2007
School of Information Technology & Engineering
University of Ottawa
Ottawa, Ontario, Canada

Your file Votre référence
ISBN: 978-0-494-49178-2
Our file Notre référence
ISBN: 978-0-494-49178-2

To my dear parents
and all the ones that I love

ABSTRACT

Software plays a significant role in society. It penetrates every field such as telecommunications, public administration, cooperation management, etc. In a software development life cycle (abbr., SDLC), software testing is a key phase. It accounts for a large part of software development costs. This is a consequence of testing being performed late in an improvised and impromptu way under the discretion of project managers. Continual testing as proposed by the extreme programming approach advocates that automated testing be performed in the early phases of the SDLC. Automation helps avoid disorderly and unsystematic progress of testing tasks and assignments. This thesis presents an approach for the generation of test cases from use cases – a form of requirements used in the early phases of the SDLC.

We first needed a way to combine related use cases in order to infer system-level test cases spawning over several use cases. We developed an approach to infer use case sequential relations based on a comparison of pre-conditions and post-conditions. This approach offers the benefit of obtaining use case sequential relations without solely relying on the traditional UML use case relationships (i.e., include, extend and generalization). It helps to avoid the functional decomposition of use cases.

We then propose an automated approach for the generation of test scenarios, a step toward complete and concrete test cases. Test scenarios are generated using depth-first traversal of control flow-based state machines obtained from use cases. The construction of these control flow-based state machines considers traditional UML use case relationships as well as inferred sequential relations. Depth-first traversal of control flow-based state machines is controlled by a coverage criterion inspired from traditional white-box code coverage.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor Dr. Stéphane S. Somé who has provided guidance and support during this research, and improved the contents of this thesis and its presentation. I am very grateful for his financial assistance towards my studies. I also appreciate my interactions with him, which initiated many inspirations from the communication with an encouraging supervisor.

My family continuously provides me with support both financially and spiritually. Without them, I would not have this chance to pursue a graduate degree from the University of Ottawa. My mother and father continuously encourage me and I have learnt to endure pain and adversity. Without their guidance, I would not have come this far and I am in deeply grateful to them for all their support. I would also like to thank to my love, who endures the long distance love and keeps encouraging me without any complaints.

It would be incomplete if I did not mention my professors during my undergraduate studies at Shanghai University, People's Republic of China. I must thank the Associate Dean of the Department of Computer Science, Dr. Gengfeng Wu, and my first supervisor Liu Yue for her tremendous support and encouragement of going abroad.

Finally, yet importantly, I must not forget all my friends who were a constant encouragement for me during good and bad times in the period of my graduate studies. I should thank specially Dajie Zhou and Wei Qiu for their support in both work and life. In addition, I thank you, the University of Ottawa, for providing me with the great opportunity of doing the Master degree in Computer Science. It was one of my aspirations and a timely decision in life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 - INTRODUCTION

1.1 Introduction

A large part of the software development life cycle (abbr., SDLC) problems is related to the testing activity. According to different authors [Kra90] [Hou98] [Kor90], the total cost of software testing can range from 40% to 55% of software development costs. Therefore, software quality is very important.

Software development and software testing are conducted under the discretion of a project manager. In order to achieve cost and schedule goals, discretion is often improvised and impromptu. This makes a SDLC redundant and tedious. Software testing should be done thoroughly within a time limit especially for critical software products such as life-critical, aviation software and critical activities of financial companies. Improvised discretions and intense time pressure eventually result in huge costs for software testing.

An approach such as extreme programming drastically shortens development cycles because it proposes continual testing throughout the process of software development [Ext97]. Continual testing inspires us to think that the testing phase can be started before the end of software development. If we use a scale as a metaphor, part of the testing weight is moved to the early phase of software development. In that case, testers may do their testing tasks without immense time pressure. According to Boehm [Boe81], a major source of expensive errors comes from the early phase of software development – the requirements. Therefore, early testing will save more, both in time and in cost, in developing a software product.

In addition to moving testing weight to the requirement phase, there are two ways to reduce testing effort. One way is to use testing tools that can execute tests automatically [Hüb03]. A deficiency of this method is that test cases are still designed manually by testers. In the case of a small system, the manual design requires testers to think about the usage and the behaviour of the system [Bec02]. However, in the case of a large system, testers must think rigorously to make overall plans and consider all factors. Moreover, the design of test cases is likely to be flawed. Another way of reducing testing effort is to generate complete and reliable test cases automatically in the early phase

of software development. Developers can build test plans earlier and obtain accurate estimates of testing costs and schedules.

1.2 Research Objective and Thesis Roadmap

In this thesis, the research objective is to propose an approach to realize the idea of early testing: generating test cases from requirements. Moreover, we adopt the second way of reducing testing effort, which is trying to generate test scenarios automatically.

Requirements are written in textual form. However, it is difficult for computers to understand informal text-style descriptions of requirements. Based on this concern, we consider using UML artifacts to formalize them. UML artifacts are mainly used in modeling, and they are widely used in the requirements analysis phase of software development [OMG99]. According to Becker [Bec02], model-based testing is based on a premise that lowering costs and improving software reliability require a tight link between functional specifications and test cases. He also mentions that testing can ensure that more attention is paid to keep models complete and up-to-date.

UML use case models are widely used to capture and model text-formatted requirements from customers. We will generate test cases from the requirement phase depicted by use cases. Use cases are used to denote functional specifications of system requirements. Test cases that are derived from use cases take advantage of the specification to ensure good functional test coverage of a system. Test case generation from non-functional requirements is outside of the scope of this thesis because they cannot be derived from use cases alone. The approach proposed in this thesis is based on use case diagrams and use case descriptions. They are two complementary instruments of use case modeling. Cockburn suggests that use cases should be written using a restricted form of natural language to ensure consistency and facilitate mechanical processing of use cases [Coc01]. In order to automate the test case generation process, we employ Cockburn's use case description and parse it into a concrete syntax. However, it is currently infeasible to fully automate the test case generation process in the requirement phase. Test cases prepared for performance testing, stress testing and other non-functional requirement testing can be achieved only after we have a very detailed design or even an implementation. Nevertheless, trying to automate as much as possible is our goal.

In order to generate system-level test cases, we must consider use case relationships besides "include", "extend" and "generalization", three common relationships in UML standard specifications. For instance, we have a common sense that the use case: "Place Order" should be executed before the use case: "Cancel Order", but there are no such include, extend and generalization relationships between the two use cases (see Figure 2-1 in section 2.3.1).

Finally, we implement our approach and integrate it into UCEd (Use Case Editor), a tool for use case modeling and requirement engineering.

1.3 Thesis Contributions and Benefits

In the context of case studies of common systems as well as UCEd, this thesis presents three contributions.

- ***First Contribution: Extraction of implicit use case sequential relations***

  We analyze dependencies between different use cases that are not bound by UML use case relationships such as include, extend and generalization and infer implicit sequential relations assumed by pre-conditions and post-conditions.

- ***Second Contribution: Test scenarios are automatically generated from control flow-based state machines with some test coverage.***

  A control flow-based state machine formalizes and integrates a set of related use cases. It is appropriate to use at an earlier software development phase. Test scenarios are generated directly and automatically from control flow-based state machines according to some test coverage. These generated test scenarios can be further applied to create complete test cases.

- ***Third Contribution: Automated scenario generation and simulation in the UCEd approach***

  Scenarios are used in the UCEd approach to validate use case requirements using simulation. We generate UCEd scenarios automatically from our test scenarios. This allows a better coverage of use cases during the validation.

We expect our work to provide the following benefits:

- Test cases will be generated with high efficiency and quality by combining detail test data to test scenarios.
- Detailed information will be provided for requirement engineering and requirement validation based on use cases.
- With the guide from the early-generated test cases, developers will be encouraged toward more in-depth modeling and precise maintenance of models of detail designs.

1.4 Thesis Outline

The chapters of this thesis cover the following issues:

Chapter 2: Context of work

This chapter introduces basic concepts and some notations. We first introduce use cases and use case scenarios. Then, we introduce use case diagrams and UML use case relationships, followed by a limitation of these relationships. Thereafter, we introduce use case descriptions and use case writing guidelines that we adopt in our approach. We will then discuss the reason to adopt use cases and state machines to generate test scenarios. In addition, we give a brief overview of test scenarios and test case generation adequacy criteria. Finally, we propose where our approach lies in the SDLC.

Chapter 3: Literature review and related work

This chapter reviews some related work that has been carried out by other researchers. We compare them with our approach.

Chapter 4: Use Case Editor (UCEd) framework and limitation

This chapter introduces UCEd. It is composed of use case model, state model, domain model and scenario model. We especially introduce how UCEd handles use case sequencing. Our implementation will add value and enrich UCEd.

Chapter 5: Inference of use case sequential relations

In this chapter, we propose an approach to capturing implicit use case sequential relations other than include, extend and generalization in use case modeling. This approach enhances test case generation on the system level. A fully automated tool is presented by applying our approach.

Chapter 6: Test scenarios generation from use cases

In this chapter, we present an approach to automatically generating test scenarios from control flow-based state machines. The generation is based on coverage criteria inspired from traditional white-box code coverage criteria. The test scenarios are automatically stored in scenario models. They can be used to further generate test cases.

Chapter 7: Case study – Team Management System

A Team Management System case study is presented in this chapter. It introduces the whole process of the system-level test scenario generation. This case study is also used for analyzing the performance of our tool.

Chapter 8: Conclusion and future work

This chapter recalls the main contributions of the thesis, limitations, and some directions for future research.

Chapter 2 – CONTEXT OF WORK

2.1 Introduction

The approach presented in this thesis will focus on how to generate test cases from formalized use cases. This approach involves much background knowledge. To make this thesis self-contained, some basic concepts and notations will be introduced.

In this chapter, we first review use cases and scenarios. The two terms are closely related, yet we need to clarify them. Then, we review use case diagrams and three common UML use case relationships. A point of view regarding a limitation in UML use case relationships is presented. After introducing use case model, we will review Cockburn's use case descriptions and propose our use case description writing guidelines. Subsequently, we will introduce test case design approaches. We also review several kinds of white-box test coverage criteria from the aspect of control-flow graphs. Finally, we introduce software development life cycles (abbr., SDLC) and propose how our approach improves the SDLC.

2.2 Use Cases and Scenarios

In some research, scenarios and use cases are regarded as synonyms. For instance, Ryser et al. simply define "use case" as "scenario" [Rys99]. We take the point of view that the two terminologies are closely related but different.

2.2.1 Use Cases

A variety of definitions of use cases have been proposed in the past fifteen years. Some of these definitions are as follows:

"A use case is a narrative document that describes the sequence of events of an actor (an external agent) using a system to complete a process." [Jac92]

"They are stories or cases of using a system. Use cases are not exactly requirements or functional specifications, but they illustrate and imply requirements in the stories they tell...domain processes

can be expressed in use cases' narrative descriptions of domain processes in a structure prose format." [Lar05]

"The specification of sequence of actions, including variant sequences and error sequences that a system, subsystem or class can perform by interacting with outside actors." [OMG99]

"A use case is a sequence of actions that an actor performs in conjunction with a system to achieve a particular goal." [Wan04]

Use cases are a means to specify the required usages of a system. They are used to capture system requirements – that is, what a system is supposed to do. System requirements are stakeholder concerns. Use-case technique models the stakeholder concerns by capturing interactions between users and a system. In addition, it separates the stakeholder concerns by applying one or more use cases. Each use case specifies a unit of useful functionality. Thus, specifying use cases follows very much the idea of building a system incrementally. In incremental software development, core functionalities (i.e., core use cases) are built before adding more advanced capabilities (i.e., more use cases).

According to Jacobson, use cases are advantageous in driving an entire system development [Jac05].

- The behaviour of a system is described step-by-step by use cases from a user's perspective, which helps to describe stakeholder concerns.
- Use cases help in planning, because critical use cases can be determined and implemented first.
- Use cases thread through classes and drive how classes work together.
- Use cases serve very well as preliminary test cases, because they describe the behaviour of the system. They provide early validation of acceptance criteria.

2.2.2 Use Case Scenarios

According to Jacobson [Jac99] [Jac05], each possible execution path in a use case is a use case scenario. A use case scenario is made up of one or more use-case events. Different use cases may have interwoven events. At a system level, a scenario may take specific execution paths through different use cases.

Two types of scenarios can be distinguished [Coc95]:

- **Main Success scenarios:** When describing a use case, we start with the simplest or the most typical scenario and trace through the flow of events. We call this a main success scenario. When executing the main success scenario, the goal of the use case is achieved.
- **Alternative scenarios:** These can be further categorized into "recovery scenarios" and "failure scenarios". The difference between a recovery scenario and a failure scenario is that the recovery scenario can ultimately recover the failure and then go back to the main success scenario to achieve the use case goal. The failure scenario, however, cannot achieve the use case goal and it ends in a failed state. In Figure 2-11, the corresponding steps in a normal procedure compose the main success scenario and the corresponding steps in the alternatives compose the alternative scenarios.

A use case contains many scenarios, which means an actor can initiate a variety of scenarios. Both main success scenarios and alternative scenarios may occur.

2.3 UML Use Case Models

A use case model contains a set of use cases related to a system. One use case model is normally composed of a use case diagram along with use case descriptions. These descriptions are usually based on use case templates.

2.3.1 Use Case Diagrams

A use case diagram is a graphical depiction of a use case model. Different use cases are depicted for a system on a use case diagram. A use case diagram consists of use cases and actors relevant to a target system and their relationships. An ellipse represents a use case and a stick figure represents an actor. A rectangle encircling the use cases represents the system (sometimes the rectangle is omitted). Lines between actor use-case pairs indicate that the actor can interact with the associated use cases [OMG03]. Figure 2-1 shows an example of a use case diagram. The system under consideration is an Order Processing System (abbr., OPSystem), which is mainly used for processing orders. There are two actors, actor "Sales Clerk", and actor "Customer", and thirteen use cases(i.e., "Turn ON System", "Customer login", "Process Order", "Place Order", "Cancel Order", "Update Order", "Sales Clerk Login", "Request Catalog", "Credit Card Payment", "Cheque Payment", "Arrange

Payment", "Update Account") in the use case diagram. A use case diagram specifies who participates in which of the system tasks. For instance, a customer can turn on a system and a sales clerk can login the system or process orders.

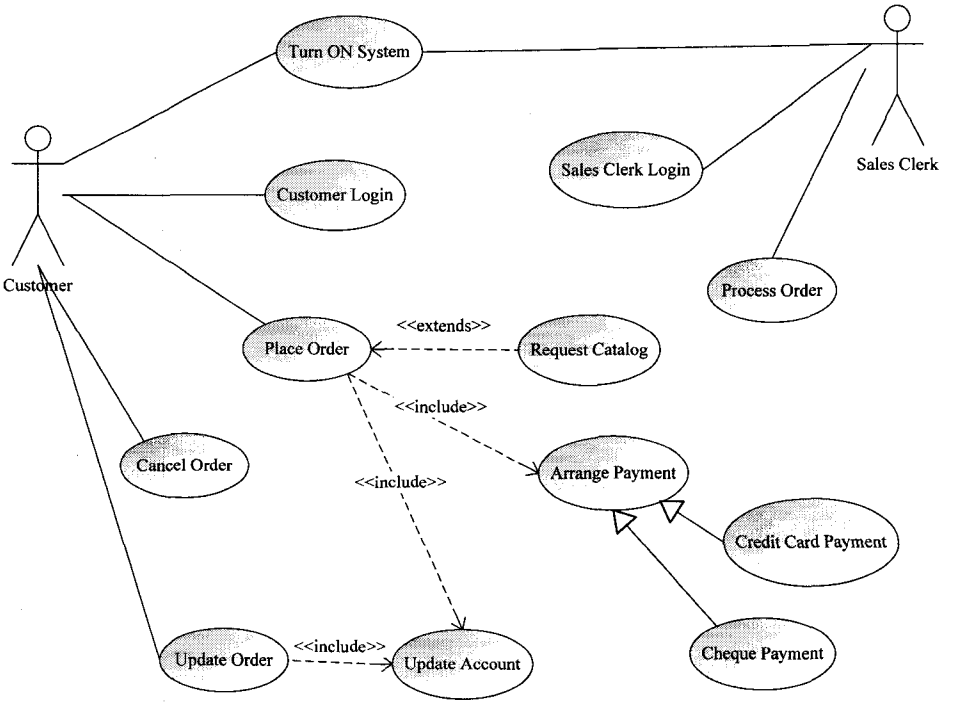
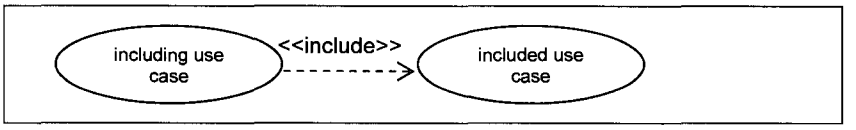


Figure 2-1: Use Case Diagram of an Order Processing System (OPSystem)

2.3.2 Use Case Relationship Analysis

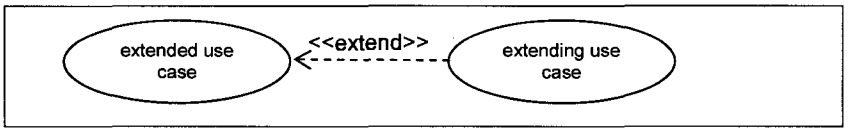
The UML defines three kinds of relationships between use cases: extend, include and generalization. Below are the terminologies needed to describe these relationships.

- **included use case and including use case:** In the presence of an include relationship, a dash arrow that is labeled with the keyword <<include>> connects two use cases. The use case that is pointed to by the arrowhead is called an included use case. The use case that is connected to the arrow-tail is called an including use case. The included use case is also called an inclusion use case.

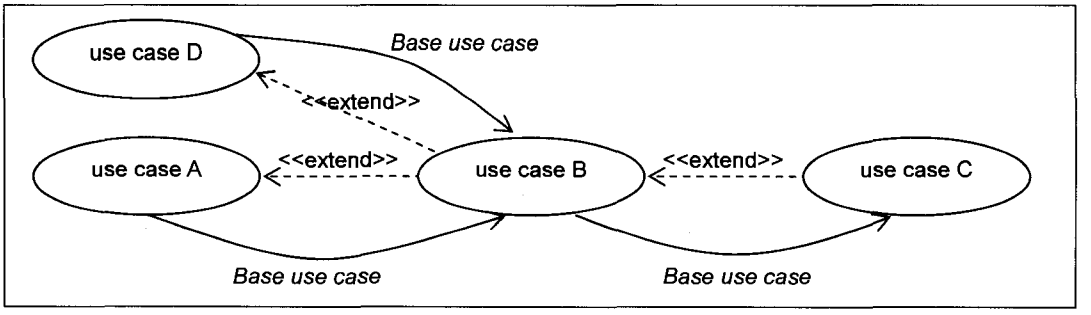


- **extended use case and extending use case:** In the presence of an extend relationship, a dash arrow that is labeled with the keyword <<extend>> connects two use cases. The use case that

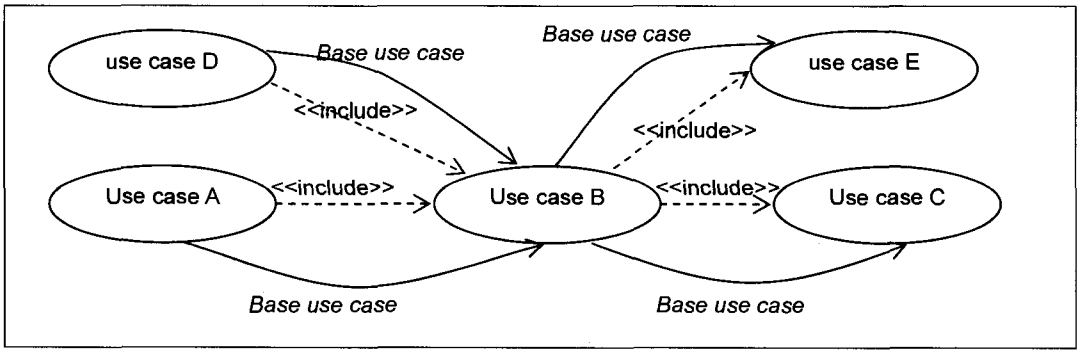
is pointed to by the arrowhead is called an extended use case. The use case that is connected to the arrow tail is called an extending use case. The extending use case is also called an extension use case.



- **Base use case:** In the presence of more than one extend relationships, the same extending use case (e.g., use case B) can extend more than one use case (e.g., use case D and use case A). Furthermore, an extending use case (e.g., use case B) may itself be extended [OMG05]. The use case that is extended is called a base use case. For instance, use case D and use case A are the base use cases of use case B. Use case B is the base use case of use case C.



In the presence of more than one include relationships, the same including use case (e.g., use case B) can include more than one use case (e.g., use case E and use case C). The same included use case (e.g., use case B) can be included in more than one use case (e.g., use case A and use case D). Furthermore, an including use case (e.g., use case B) may itself be included. The use case that includes other use cases is called a base use case. For instance, use case D and use case A are the base use cases of use case B. Use case B is the base use case of use case E and use case C). Therefore, base use case is a relative definition.



- **Normal use case:** A normal use case is opposite to an extending use case (extension use case). A normal use case is the use case that does not extend other use cases. Including use cases and included use cases are normal use cases.
- **Fragment use case:** A fragment use case is opposite to a normal use case. An extending use case (extension use case) is regarded as a fragment use case.

For instance, in the OPSystem use case model presented in Figure 2-1, the use case: "Request Catalog" is a fragment use case. It is also an extending use case (extension use case) that extends the use case: "Place Order." The use case: "Place Order" (an extended use case) is the base use case of the use case: "Request Catalog". The use case: "Place Order" is also an including use case and a base use case of the use case "Update Account" (an included use case). We will now introduce include, extend and generalization in detail.

- **Include Relationship**

  Certain steps are similar across different use cases. These common flows of events can be factored into what is termed an included use case. Other use cases can refer to the flows within the included use case. The reference from an including use case (a normal use case) to an included use case (a normal use case) is modeled with an include relationship. It also describes that the behaviour of an including use case contains that of another included use case [Iso05] [Jac05]. Figure 2-2 presents a control flow of an include relationship between the use case: "Place Order" and the use case: "Update Account". Figure 2-3 presents the courses of the use case: "Place Order" and the use case: "Update Account". The execution path begins at step *1* of the use case: "Place Order", continues (i.e., in step *5*) through an included flow within the use case "Update Account", and resumes at the use case: "Place Order".

- **Extend Relationship**

  An extending use case (an extension use case) continues the course of an extended use case by conceptually adding new behaviours into its extended use case [Jac05]. The meaning of the extended use case is independent of the extending use case. The behaviour defined in the extending use case may not necessarily be meaningful by itself [OMG05]. An extend relationships specifies that the flows of an extending use case may be added into its extended

use case depending on an extension condition. Figure 2-4 presents control flows of an extend relationship. Figure 2-5 presents the courses of the use case: "Place Order" and its extension use case: "Request Catalog". An actor instance triggers the use case: "Place Order" and a use-case instance of the use case: "Place Order" is created. It goes through the steps until step *1*. If the extension condition "User requests catalog and catalog is available" is satisfied, the extension flow in the use case: "Request Catalog" is executed. Then, the extension flow resumes the flow in the use case: "Place Order". In the last step, the use-case instance is terminated. On the other hand, if the extension condition is not satisfied, no extension takes place so that the use case: "Request Catalog" will not execute.

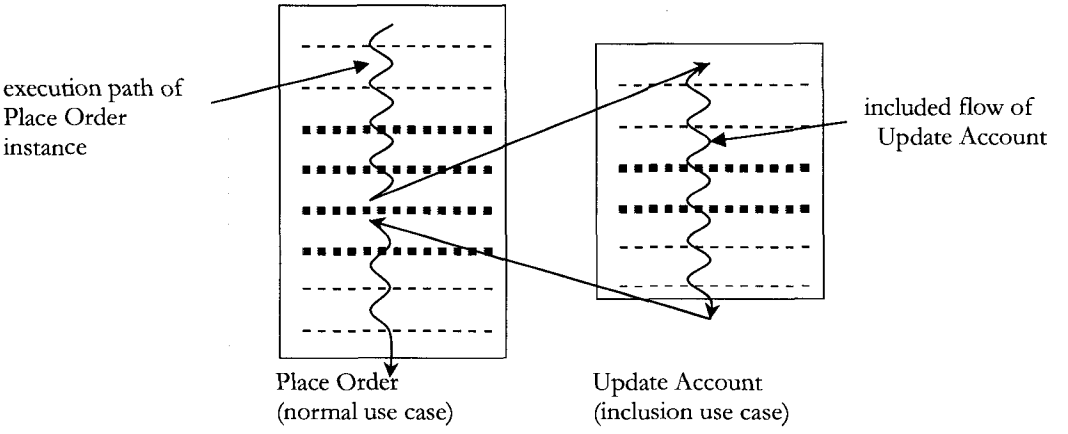


Figure 2-2: Control flow of an include relationship

[Basic course of use case: "Place Order"]

1: Customer selects items
2: System identifies item and put it to the shopping cart
3: Customer types the number of the items
4: System confirms the credit.
5: Include Update Account
[use case: "Update account"]

Basic course:
1: Customer account information is listed.
2: Customer changes the account information and confirm
3: System verification.
4: System displays successfully submitted information.
Alternatives:
3a If format or typo occurs.
3a1. Error message displays.
3a2. Go back to Step 2.

Figure 2-3: Use case example of an include relationship (use case: "Place Order" and use case: "Update Account")

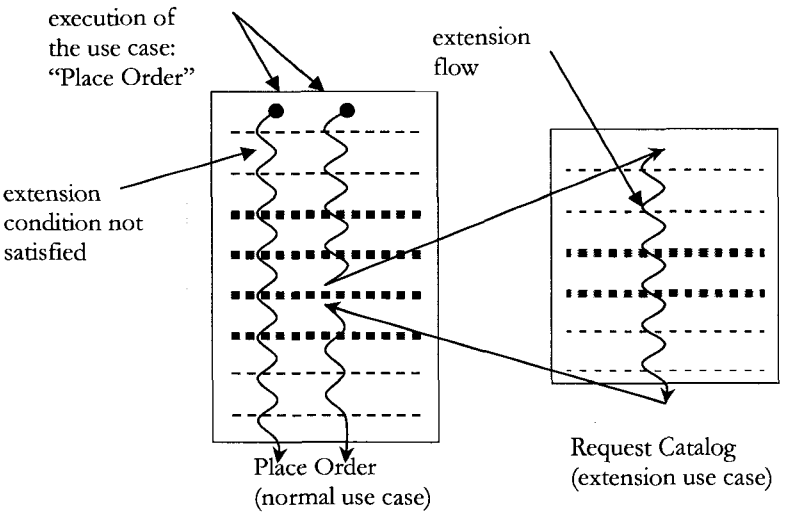


Figure 2-4: Control flow of an extend relationship

[Basic course of use case: "Place Order"]

1: Customer selects items
2: System identifies item and put it to the shopping cart
3: Customer types the number of the items
4: System confirms the credit.
5. Include Update Account
[use case: "Request Catalog"]

Extension condition: In step 1, user requests a catalog for the selected items and Request Catalog available

Basic course:

1: System mails requested catalog to the mailing address.
2: System displays successful message.

Figure 2-5: Use case example of an extend relationship (use case: "Place Order" and use case: "Request Catalog")

- **Generalization relationship**

A group of use cases may have similar flows of events or similar set of constraints that are imposed on them. These flows of events can be generalized and described in their parent use cases, rather than specifying them and constraints repeatedly in individual use cases [Jac92] [Jac99] [Jac05]. Generalization can denote an "is-a-kind-of" relationship between use cases [OMG03]. It factors out the common behaviour into a parent use case and puts specific behaviour in a child use case. The parent use case has some flows that have been identified but are deliberately unspecified; these flows are considered abstract. Actual specifications of these abstract flows are postponed to child use cases. In UML, we say that the child use cases make the flows concrete by providing the specifications for these flows [Jac05]. In the OPSystem

example, there are different kinds of payments. Customers can make payments by credit card or by cheque. Payments are generalized as the use case: "Arrange Payment" (see Figure 2-1). Figure 2-6 presents the control flow of a generalization relationship. Figure 2-7 presents the courses of the general use case (i.e., a parent use case): "Arrange Payment" and the special use case (i.e., a child use case): "Credit Card Payment". The instantiation occurs at the child use case: "Credit Card Payment". An execution path of the child use-case instance follows through a number of flows. If there is no flow defined in the child use case, the flow defined in the parent use case is executed. The child use-case instance follows the basic flow of the use case: "Arrange Payment". This is because the use case: "Credit Card Payment" does not define it, but inherits from its parent. When the use-case instance reaches the point to choose a payment method, we find that the parent use case leaves it abstract (step *2* of the use case: "Arrange Payment"). Thus, the flow of the use case: "Credit Card Payment" is executed.

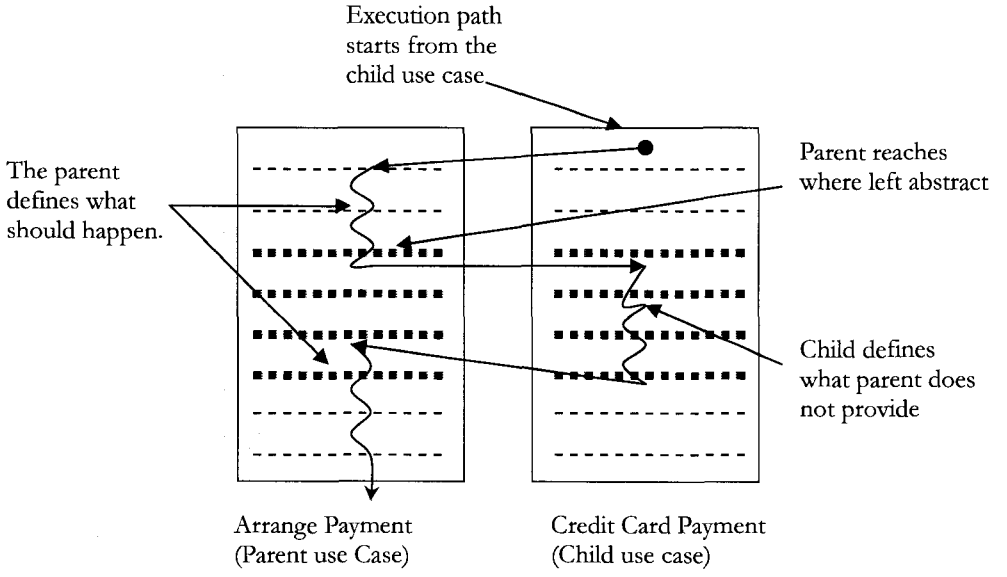


Figure 2-6: Control flow of a generalization relationship

[Basic course of use case: "Arrange Payment"]

Basic course:
A1: System displays Payment method selection
A2: Customer selects Payment Method.
A3: System displays order number and record
A4: System displays Payment complete
Sub-course:
{abstract} Payment Method

[use case: "Credit Card Payment"]

Basic course:
...
Sub-course: Payment Method
B1: Customer selects Credit Card Payment
B2: Customer types credit card number
B3: Customer types expiration date
B4: System validates credit card information

[use case: "Cheque Payment"]
Basic course:
...
Sub-course: Payment Method
C1: Customer selects Cheque Payment
C2: System puts order on hold.

Figure 2-7: Use case example of generalization relationships (use case: "Arrange Payment" and use case: "Credit Card Payment", use case: "Cheque Payment")

2.3.3 A Limitation of UML Use Case Relationships

In the previous section, we introduce the three UML use case relationships from the viewpoint of use-case flows. These relationships describe interdependencies among use-case flows. Once a use case is instantiated, the use-case instance will follow an execution path until the use-case instance is terminated. With the three relationships, this execution path can traverse different use cases. However, there are no provisions in UML specification to describe sequential relations [Bin99] among use-case flows. UML use case relationships do not allow the expression of the fact that a use-case flow would be followed by another use case. For instance, after the use case: "Customer Login" is successfully completed, the use case: "Place Order" can be instantiated by an actor: "Customer" as well. The use-case flow in the use case: "Place Order" can continue the path in the use case: "Customer Login". In order to lead use cases naturally and completely to the identification of test cases, the sequential relations (e.g., the use case: "Customer Login" is followed by the use case: "Place Order") among use-case flows should not be omitted.

Moreover, we should avoid abusing UML use case relationships. According to Fowler[Fow98], too much functional decomposition by breaking a use case down to sub-use cases will cause trouble in several ways:

- Use cases are external structures. The people who build them and do the functional decomposition are usually the ones who lack object-oriented experience. It is not necessary to make the external structure look like an internal structure of a system. Most of the benefits of objects will be lost and behaviours across these objects will be duplicated.
- It will be difficult to use the same behaviour in another context, even if it is the same. This is because functional decomposition requires us to think at a high level. Putting too much effort on structuring use cases will dig into an overly low level.
- It takes a very long time to capture every detail of use cases in the early phase of development. In addition, arguments will occur on deciding which use case will fit into which higher-level use case.

Due to the situation that less UML use case relationships should be used to avoid too much functional decomposition, it is more natural and important to use sequential relations - implicit interdependencies among different use cases. The use cases that have sequential relations are on the same level. To capture the sequential relations does not decompose the functions of the use cases but their use-case flows can be combined. The sequential relations are crucial in testing; they stand central to complete testing. This means that no test can yield satisfactory results if the sequential relations between use cases are not considered and tested for. More details of this approach will be introduced in Chapter 5.

2.4 Use Case Descriptions (Use Case Specifications)

Use case diagrams are a graphical depiction for requirements elicitation. However, the information that they convey is not enough. Use case diagrams represent actors, use cases and their communications, but they do not show us how actors communicate with use cases; neither do we know the content of use-case flows. Figures 2-3, 2-5 and 2-7 have already presented more detail about use cases. They are called use case descriptions (also called use case specifications). In this section, we describe the technique for writing use cases in Cockburn's format. Afterwards, we elaborate on our guidelines for writing use case descriptions.

2.4.1 Cockburn Use Cases

A. Cockburn proposed a semi-formal use case description, which is a table-form use case template to express behavioural information of use cases. It is a goal-oriented form. Figure 2-8 presents the table template recommended by Cockburn [Coc01].

Use Case Name	The name of the use case <the name is the goal as a short active verb phrase	
Goal in Context	<A longer statement of the goal in context if needed>	
Scope & Level	<what system is being considered black box under design> <one of: Summary, Primary Task, Subfunction>	
Pre-conditions	<what we expect is already the state of the world>	
Success End Condition	<the state of the world upon successful completion>	
Failed End Condition	<the state of the world if goal abandoned>	
Primary, Secondary Actors	<a role name or description for the primary actor> <other systems relied upon to accomplish use case>	
Trigger	<the action upon the system that starts the use case>	
Main Success Scenario	**Step**	**Action**
	1	<put here the steps of the scenario from trigger to goal delivery, and any cleanup after>
	2	<…>
Extensions	**Step**	**Branching Action**
	1a	<condition causing branching>: <action or name of subordinate use case>
Sub-variations		**Branching Action**
	1	<list of variations>

RELATED INFORMATION	<Use case name>
Priority	<how critical to your system/organization>
Performance	<the amount of time this use case should take>
Frequency	<how often it is expected to happen>
Channels to actors	<e.g. interactive, static files, database, timeouts>
OPEN ISSUES	<list of issues awaiting decision affecting this use case
Due Date	<date or release needed>
…any other management information	<…as needed>
Superordinates	<optional, name of use case(s) that includes this one>
Subordinates	<optional, depending on tools, links to subordinate use case>

Figure 2-8: The use-case table template proposed by A. Cockburn

Goal in context should be structured around a business case. *Scope* includes functional scope required to achieve the goal. *Level* can be divided into three kinds: summary goals, user goals (primary tasks) and sub-functions. Figure 2-9 [Coc01] describes the three different goal levels by drawing a sailboat. The sail is a summary goal; the sailboat body on the seabed is user goals and the sailboat body under the seabed is sub-functions. User goals are the goals of greatest interest for primary actors to get work done. They reflect elementary business processes. Summary goals represent a collection of

user goals to provide an index of a large set of user goals. Sub-functions are steps below the main level of user interests. Cockburn indicates that such depiction of goal levels reflects from high-level down to low-level use cases. The higher the use cases are, the higher level and longer range they are. The lower the use cases are, the lower level and shorter range they will be. Summary goals-level use cases normally require completion of the use cases located below them which could be user-goals-level use cases or even sub-functions goals-level use cases [Coc01].

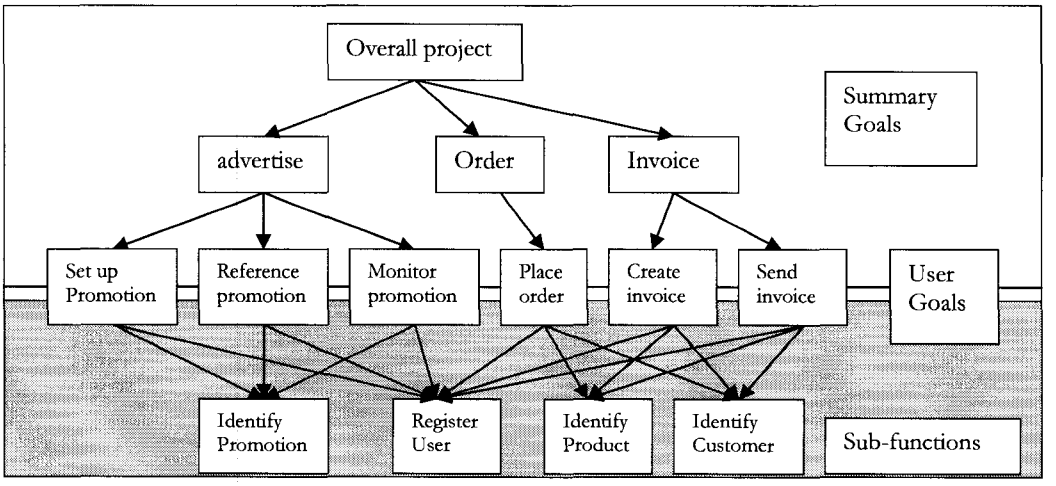


Figure 2-9: The level of use cases. The use case set reveals a hierarchy of goals

Pre-conditions are prerequisites before the execution path created by the use-case instance can be started. *Post-conditions* are divided into success-end conditions and failed-end conditions. The success-end condition is considered a successful end to the execution path. The failed-end condition is considered a failed end to the execution path. *Primary actors* are actors who use a system to achieve a goal and instantiate the user cases. Common primary actors could be customers, clerks, etc. *Secondary actors* are actors from whom a system needs assistance to achieve the primary actors' goal. Common secondary actors could be an order system, a client management system, etc. *Trigger* is the action upon the system that starts the use case. *Main success scenario* contains several steps, describing the main flow of events to achieve the success goal. *Extensions* are events altering from the main success scenario. Each step in extensions refers to a step in the main success scenario. *Sub-variations* are eventual bifurcations that can be regarded as an attempt to make them more distinct from extensions. Typical sub-variations are "buyer may pay by cash", "buyer may pay by money order" or "buyer may pay by credit card". Priority can be set as *Low*, *Medium*, *High* or *Top* that helps to better manage direction of development. *Performance* can vary from different systems. *Channels to Actors* [Coc01] should indicate how the actors interact with a system, such as UI for human actor, ODBC

or API for database or third party system external actors.

2.4.2 Our Guidelines of Writing Use Cases

Free-form natural language suffers from an inherent ambiguity. A compromise between formality and understandability is to use a restricted form of natural language. In this thesis, use case descriptions are based on Cockburn's use case descriptions with some improvement.

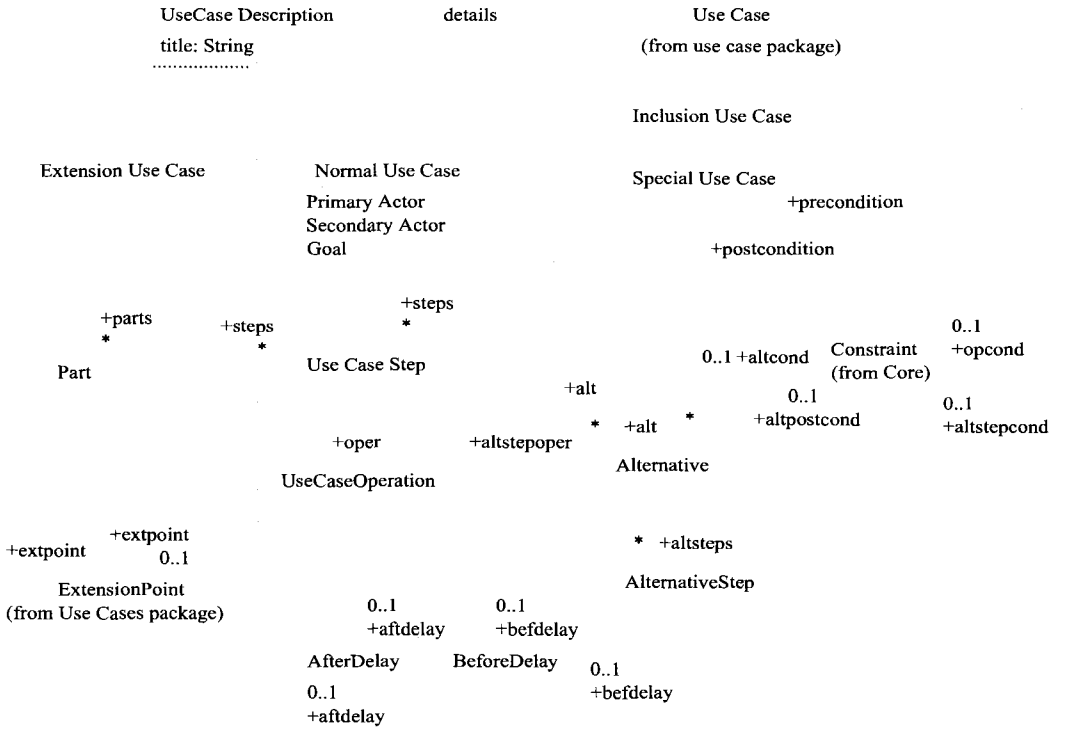


Figure 2-10: The UML representation of an abstract syntax for use case descriptions

We use a formalization to write use cases. It adheres to an abstract syntax [Som03] [Som04b] [Som05b] based on UML use-case definition. Figure 2-10 shows an improved abstract syntax. A sign "0..1" means that a component can either not appear or appear once. A sign "*" means that a component can appear zero or more times. If there is no specific sign, it simply means that a component can appear only once. Normal use cases can be inherited to inclusion use cases in the presence of include relationships, and special use cases in the presence of generalization relationships. It may have fragment use cases. A fragment use case can be an extension use case.

Title: Place Order
Primary Actor: Customer
Secondary Actor/Participants: Order System
Goal: A customer wants to browse the shopping list and place order.
Pre-condition: Order System is ON AND Customer is logged in
Steps:
1: Customer selects items
Extension point ➔ item selected
2: System Identify item and put it to the shopping cart
3: Customer type the number of the items
Extension point ➔ number typed
4: System validates the number.
5: Include Update Account
6. Include Arrange Payment.
Alternatives:
4a Item number is invalid
4a1. Error Message
4a2 Goto Step 3
Success Post-condition: Order is placed AND Order System is ON AND Customer is logged in

Figure 2-11: Use case: "Place Order" describing a place order procedure in an OPSystem

A normal use case includes a *primary scenario* (i.e., main success scenario) and zero or more *secondary scenarios* (i.e., alternatives), which are alternative flows of events to the *primary scenario* [Sch98]. Figure 2-11 presents an example of a normal use case. The *primary scenario* is described in the section titled *Steps*. The *secondary scenario,* which is described in the section titled *Alternatives*, consists of flows of events continuing the *primary scenario*. A *normal use case* is composed of the following elements as a tuple [*UCTitle, PActor,* {*SActor*}, *UCGoal, UCPrec, UCPost, UCSteps* [{*Opcond*}, *Oper,* {*Alt* [*AltCond,* {*AftDelay*}, {*BefDelay*}, *AltSteps,* {*AltPost-condition*}]}, {*UCAlt*}, {*ExtPoint*}]]. The elements in "{}" may or may not appear. *UCTitle* is a label that uniquely identifies the use case. Normally a title is the use-case name. The primary actor (*PActor*) is the actor that plays an active role in the normal use case. The secondary actor (*SActor*) is a participant that plays a passive or secondary role in the normal use case. The *UCGoal* is a statement of the objective or the primary actor's expectation upon a successful completion of the use case. *UCPrec* is a condition that must be true prior to an execution of the use-case instance. *UCPost* is a condition that must be true at the end of the successful execution of the use-case instance (use case main scenario). *UCSteps* is a sequence of steps including the sequential actor's actions and system reactions. Operation (*Oper*) is a use-case operation, which can be distinguished into three types: *instances of concept operations, branching statements* (*GOTOs*) and *use case inclusion directives.*

- An *instance of concepts operation* is an event that denotes the execution of an operation by a concept (an actor in the environment of the system or the system itself). It includes triggers (actor's actions) and system reactions. The instances of concept operations in the use case: "Place Order" include "selects items" by concept "Customer" (i.e., an actor) and "identifies items" by concept "Order Process System" (i.e., the system under consideration).
- A *branching statement* refers to a target step to which the execution path continues. Target steps are restricted to the steps in the primary scenario. Branching statements can only appear in alternatives, because we assume that any jumping steps are not allowed in the primary scenario. For instance in Figure 2-11, step *4a2* in the use case: "Place Order" continues to step *3*. Therefore, the execution path would result in continuing the flow of events from step *3*.
- A *use case inclusion directive* is a realization of an include relationship between an included use case and an inclusion use case that is referred to by the directive.

An operation may include a guard (*OpCond*) and a timing condition. The guard must be hold for the operation to be possible. The timing condition can be distinguished to an *After Delay* and a *Before Delay*. An *After Delay* specifies the minimum period that must pass for the operation to be possible; a *Before Delay* specifies the maximum period after which an operation is no longer possible. *UCAlt* is a set of alternatives that apply to all the steps in the use cases. Alternatives (*Alt*) are a set of alternatives that may apply to each step in the use case. A step can have zero or more alternatives specifying exceptional behaviours. An alternative (*UCAlt*) describes an exceptional flow of events in the use case. It specifies a possible continuation after a specific use-case step. Alternative condition (*AltCond*) is a condition that must be true for the alternative step to be possible. An alternative step (*AltStep*) is a use-case operation, which is an instance of concept operations or a branching statement. All of the alternative steps must have an alternative condition. The use case inclusion directives cannot appear in alternative steps. An alternative post-condition (*AltPost-condition*) is a condition that must be true after the execution of alternative steps when the steps lead to a failure scenario (refer to section 2.2.2, alternative scenario). An extension point (*ExtPoint*) is a label, which is a reference to the extension point in the parts of the extension use case. A point interaction defined in the parts of the extension use case may be inserted.

Title: Request Catalog
Parts:
p1: At extension point item selected
1p1 System mail requested catalog to the mailing address.
p2: At extension number typed
1p2 System mail requested catalog to the mailing address

Figure 2-12: Extension use case: "Request Catalog" describing a Requesting Catalog procedure in an OPSystem

Figure 2-12 is an extension use case. An extension use case contains one or more *parts* which are inserted at specific *extension points* in a normal use case. An extension use case is composed of the following elements as a tuple [*UCTitle, Parts*[*ExtPoint, Steps*]]. *Parts* is a set of parts and normally includes an extension point (*ExtPoint*) and *Steps*. An extension point here is a reference to the extension point defined in normal use cases. Each extension point refers to a step in the extended use case. Steps in extension use cases can simply be considered an instance of concept operation. The extension use case may participate in its extended use case under a specific extension condition.

Besides the abstract syntax, a concrete syntax is proposed to write use-case conditions and operations [Som06]. A use-case condition is a predictive sentence that is specified by an associated domain entity. Conditions describe situations prevailing within the system. Domain entities are from stereotypes *concept attribute* or *concept* that are instances of *concept attribute*. A use-case operation describes an action initiated by a subject entity in a use case sentence. Operation may be an instance of concept operation, a branching statement, or a use case inclusion directive. Figures 2-13 and 2-14 present the concrete syntax for use-case conditions and operations [Som06].

```
<condition>   -> <acondition> "and" <condition>
            | <acondition> "or" <condition>
            | "("<condition>")"
            | <negation> <condition>
<acondition>  -> [<determinant>] <entity>[<verb>]<value>
<determinant> -> "a"| "an"| "the"
<negation>     -> "not"| "no"
<entity>       -> <concept> | <attribute>
<aggregate>    -> <word> + {member of the model concepts}
<concept>      -> <word> + {member of the model concepts}
<attribute>    -> <word> + {attribute of concept}
<verb>         -> {derived from to be in present tense}
<value>        -> <word> + {value of the entity}
                | <comparison> {entity is non-discrete ?}
<comparison>  -> <comparator> <word>
<comparator>  -> ">" | "<" | "=" | "<=" | ">=" | "<>"
               | "greater than" | "less than" | "equal to" | "different to" |
"greater or equal to" | "less or equal to"
```

Figure 2-13: Concrete syntax for use case conditions

```
<operation_spec>   ->   <concept_operation>   |   <branching_statement>   |
                    <useCase_inclusion>
<concept_operation> -> [<before_delay>] [<after_delay>]
                       [<condition_spec>] <operation_reference>
<condition_spec>-> "IF" <condition> "THEN"
<operation_reference> -> <word> + {derived from an operation of the current
                                   entity}
<after_delay> -> "AFTER" <duration_spec>
<before_delay> -> "BEFORE" <duration_spec>
<duration_value> -> <duration_value> <duration_unit>
<branching_statement> -> "GO" "TO" "Step" <word> {corresponding to a step
                                   label}
<useCase_inclusion> -> [<condition_spec>]
                       "INCLUDE" <use-case-name>
```

Figure 2-14: Concrete syntax for use case operations

2.5 Finite State Machines and StateCharts

A general finite state machine (abbr., FSM) can be defined as a tuple $[\sum, S, T, F, S_0]$ where $\sum$ is a set of events, S is a set of states, T is a set of transitions, F is a transition function and $S_0 \in S$ is an initial state. Control flow-based state machines (abbr., CFSM) are types of FSM that are particularly designed to depict the flows of events. Normally, the transitions in a CFSM have two attributes, event and condition. Sometimes, however, the event and condition can be described in states instead of being described in transitions (refer to the next section). In light of the introduction from Bruce [Bru94], the event specifies an action when a transition is traversed by a system receiving a message. The event specifies a response when a transition is traversed by a system sending out a message. Figure 2-15 presents an example of a CFSM. For instance, State *0* (S_0) is the initial state. There are ten states (S) and twelve transitions (T). In the twelve transitions, eight of them (labels without "[ ]") are events ($\sum$) and four of them are conditions (labels with "[ ]"). $(S_0 \times \text{Select items} \times S_1) \in F$ is a function.

Statecharts were originally introduced by D.Harel [Har87]. The statecharts formalism is an extension of FSM with a hierarchical structure and a concurrent mechanism of communications [Bri05]. Statecharts are reactive behaviour model descriptions where transitions represent complete interactions. Each transition may include a trigger event (an operation performed by an actor) and system reactions resulting from that event. Transitions in a state model, however, each correspond to a single event, which is a trigger event or a system reaction. Figure 2-16 presents an example of a statechart corresponding to the CFSM presented in Figure 2-15. Although the number of the states and transitions is less than that of the CFSM, the transitions contain more information that

combines conditions, actions and responses where applicable.

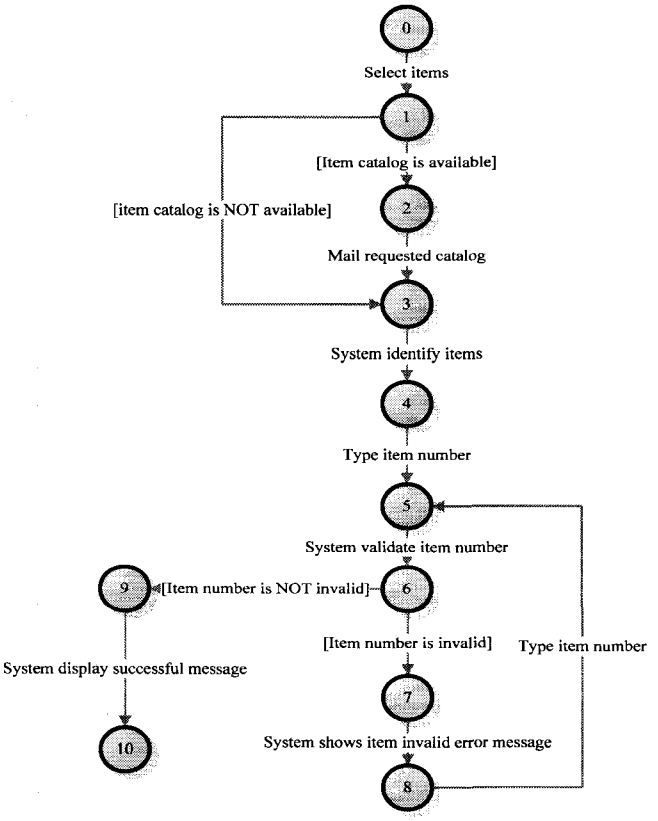


Figure 2-15: An example of a control flow-based state machine

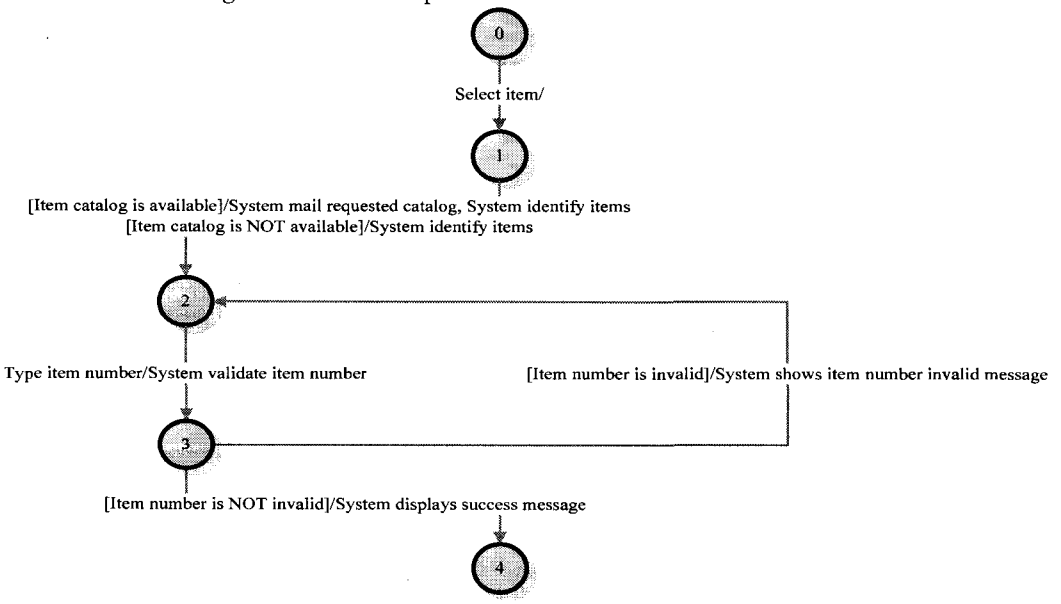


Figure 2-16: An example of a state chart

CFSM and statecharts are both state-based specifications and directed graphs. In this thesis, we use FSM rather than statecharts for test generation for the following reasons:

- FSM are easier to traverse than statecharts. We can achieve all sequences directly from FSMs by applying different traversal algorithms such as a depth-first algorithm or a breadth-first algorithm. On the contrary, statecharts must be transformed into other forms such as a testing flow graph [Kan03]. For instance, Binder [Bin99] introduces a transition tree to store the traversal result from a statechart. Each path in the tree starting from the root and ending at a leaf node may represent a test case.
- It is easier to analyze test coverage from FSMs than from statecharts, because a transition only stands for one event. In statecharts, each transition has combined events.

2.6 Test Cases

In general, a test case is a question that is asked to a program. More concretely, a test case is a document that describes an input, action or event and an expected response. According to Briand et al. [Bri02], test cases can be derived from test requirements with some detailed designs. A test case should contain a test case identifier, a test case name, a test goal, test setup, test conditions, test data inputs, test scenarios (i.e., steps) and expected results. Test conditions are the conditions that must be met before executing test cases. Expected results contain descriptions of what testers should see after all test steps have been completed. Test cases can be derived from traditional requirements or use cases. Some literatures regard test data (data value inputs plus test scenarios) as test cases [Kor90]. However, we think that test data should only include data value inputs, which can be combined into test scenarios to build concrete test cases.

2.7 Test Design Approaches

Black-box test design approaches and white-box test design approaches are commonly used in test case designs. Black box-based test design approaches are based on the system's inputs and outputs described in a specification. White box-based test design approaches are based on software's internal logic structure. When testers are conducting test case designs, the white box-based test design approaches often become available to them in a later phase of development in contrast to the earlier availability of the knowledge necessary for the black box-based test design approaches. Therefore, for a given project's progress, the white box-based test design always follows the black box-based

test design. The level of detail required for white box-based test design is very high, so testers can only consider very small granularities of the items when developing test cases. Hence, white box-based test design is more useful to test small components and black box-based test design can be used for both small and large components.

In white box-based test design approaches, control-flow graphs are widely used for coverage analysis. Coverage is a measure used for software testing that describes the degree and adequacy of which the program has been tested [Mil63]. Control-flow graphs can be used by testers to evaluate code with respect to testability, as well as to develop white-box test cases. In a black box-based test design approach, coverage is also considered, for example, to cover certain functional requirements, to cover certain equivalence classes or to cover certain system features. In contrast to black-box test design approaches, white-box test design approaches have stronger theoretical and practical support.

Besides the two test design approaches, the term grey-box testing has come into common usage in recent years [Ngu03]. The technique used in our approach can be categorized in grey-box testing for the following reasons.

- By analyzing software and reverse engineering, the white-box approach creates flow charts from code implementations and program structures can then be analyzed to generate test cases [Jor95]. Similarly, our approach will generate test cases from CFSMs. However, our approach does not access the source code since use cases are created in the requirement phase. Our approach can be used in the earlier phase of the SDLC in which the white-box approach cannot be used.
- Black-box approaches only validate whether the function specified in the system requirements specifications are implemented or not with no knowledge of the internal system [Bei95]. Our approach is beyond black-box approaches with limited knowledge of the internal system. Test cases are generated based on the information such as state-based model and diagrams of the system rather than the system requirement specifications.
- Use case specifications preserve the essential information from the requirement, and they are the basis of the code implementation. Our approach of generating test cases from use cases combining the white-box approach and the black-box approach. It employs the coverage criteria of the white-box approach and finds all the possible paths from the use cases that describe the expected behaviour of the system. Test cases that the conditions to execute the paths are

satisfied by the black-box approach are then generated.

2.8 Test Adequacy Criteria

A major goal of test strategies is to test a whole system, i.e., to test all requirements and features. Exhaustive testing is expensive and labour-intensive because of the large number of test cases that are generated. Thus, exhaustive testing is usually not feasible, and a frequently used type of testing is selective system testing. A set of test cases, meeting a coverage criterion is frequently used to test selected system functionality. Selected features of a system are tested with respect to a requirement class, e.g. performance, security, etc. A test coverage strategy may significantly reduce the number of test cases as compared to exhaustive testing while preserving the high quality of a complete test suite with respect to the selected set of requirements.

Due to the significant number of test cases that may be possible, we apply test coverage strategy according to some adequacy criteria. A control-flow graph facilitates the design of white box-based test cases as it clearly shows the logic elements needed to design the test cases using the coverage criterion of choice. For instance, (i) statements; (ii) decisions/branches; (iii) conditions (expressions that evaluate to true/false, and do not contain any other true/false-valued expressions); (iv) combination of decisions and conditions; (v) paths (node sequences in flow paths) [Bur02]. Figure 2-17 is a sample code and Figure 2-18 shows the corresponding control-flow graph. The nodes represent sequential statements, as well as decisions and looping predicates. The edges in the graph represent transfer of control, and the direction of the transfer depends on the outcome of the condition in the predicate (true or false).

```
1.  Begin
2.    While (X>=80 and Y>=80)
3.      X=X-5;
4.      Y=Y-5
5.      if (X+Y>=140 and (X>=90 or Y>=90)) then
6.        Sum [i] = X + Y;
        endif
7.    i=i+1;
      endwhile
8.  End
```

Figure 2-17: Code sample with branches and loops

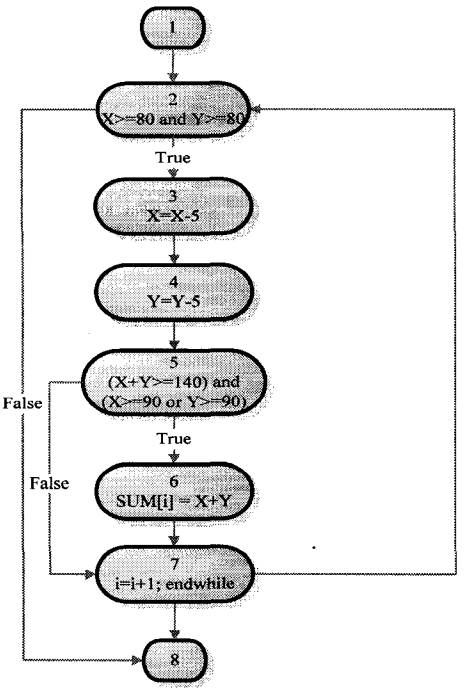


Figure 2-18: A control-flow graph representation of the code in Figure 2-17

2.8.1 Statement Coverage Criterion

If we call the code sample presented in Figure 2-17 a module, for statement coverage criterion, all of the statements in the module are executed at least once. There are eight statements in the module. In terms of a control-flow graph of the code, achieving complete (100%) statement coverage requires that all nodes in the graph be exercised at least once by test cases. For the control-flow graph in Figure 2-18, a tester will have to develop test cases that exercise nodes *1* to *8*. The test case presented in Table 2-1 satisfies the statement adequacy coverage.

ID	X	Y	Path
1	95	84	1-2-3-4-5-6-7-2-8

Table 2-1: Test case that satisfies the statement coverage criterion

2.8.2 Branch/Decision Coverage Criterion

To achieve complete (100%) branch/decision coverage, each decision element in the code (loop, case, if-then, etc.) should be executed at least once. In terms of the control flow graph, this requires that all the edges in the corresponding flow graph be exercised at least once. Complete branch coverage is considered a stronger coverage goal than statement coverage. This is because covering all

the edges in a control flow graph will ensure all the nodes are covered. The test cases presented in Table 2-2 satisfy the branch coverage criterion.

ID	X	Y	Path
2	95	84	1-2-3-4-5-6-7-2-8
3	94	84	1-2-3-4-5-7-2-8

Table 2-2: Test cases that satisfy the branch coverage criterion

2.8.3 Condition Coverage Criterion

Decision coverage only requires that all possible outcomes for the branches or loops be exercised at least once, not for each individual condition contained in a compound predicate. A compound predicate contains multiple conditions. In decision coverage, individual conditions contained in a compound predicate are considered as a whole. This may cause some defects to go undetected. For instance, in branch coverage, the test cases would not exercise the possible outcome for X>=80 as "False". For that reason, a defect in the logical operator for predicate 1 (X>=80 and Y>=80) may not be detected. Condition coverage requires that each individual condition in a compound predicate take on all possible values at least once. For instance, the decision (if (X+Y>=140 and (X>=90 or Y>=90))) has three individual conditions in its predicate: (i) X+Y = 140, (ii) X>=90, and (iii) Y>=90. This decision has a compound predicate. Among the statements in the code sample, two of them are predicates. Both of the predicates are compound predicates.

Predicate 1 in decision (if (X>=80 and Y>=80)):

Individual condition 1: X>=80

Individual condition 2: Y>=80

Predicate 2 in decision (if (X+Y>=140 and (X>=90 or Y>=90))):

Individual condition 3: X+Y>=140

Individual condition 4: X>=90

Individual condition 5: Y>=90

ID	X	Y	C1	C2	C3	C4	C5	Path
4	94	90	True	True	True	True	True	1-2-3-4-5-7-2-7-2-3-4-5-7-2-3-4-5-7-2-8
5	70	69	False	False	False	False	False	1-2-8

Table 2-3: Test cases that satisfy the condition coverage criterion

The test cases presented in Table 2-3 satisfy the condition coverage. All of the five individual conditions take on "True" and "False" at least once. However, condition coverage cannot replace

branch/decision coverage. Not all possible outcomes for the decision are exercised so it would not meet the branch/decision coverage. The decision X+Y>=140 and (X>=90 or Y>=90) does not take on a "True" outcome (refer to the path).

2.8.4 Branch/Condition Coverage Criterion

A more stringent coverage criterion requires that every condition be set to all possible outcomes and the decision as a whole be set to all possible outcomes. This coverage criterion is called Branch/Condition (Decision/Condition) coverage. A combination of test cases (Table 2-4) in Table 2-2 and Table 2-3 would satisfy this criterion. Test case 4 is eliminated because test cases 2, 3 and 5 are able to satisfy the criterion already.

ID	X	Y	C1	C2	C3	C4	C5	Path
2	95	84	True	True	True	True	False	1-2-3-4-5-6-7-2-8
3	94	84	True	True	True	True	False	1-2-3-4-5-7-2-8
5	70	69	False	False	False	False	False	1-2-8

Table 2-4: Test cases that satisfy the branch/condition coverage criterion

2.8.5 Condition-Combination Coverage Criterion

The criteria described above do not require the coverage of all possible combinations of individual conditions. Another criterion called condition-combination adequacy coverage (also called multiple-condition coverage) requires that all possible combinations of condition outcomes in each decision must occur at least once. For instance, the decision (if (X>=80 and Y>=80)) needs to be satisfied by the following combinations:

Condition1	Condition2
True	True
True	False
False	True
False	False

The decision (if (X+Y>=140 and (X>=90 or Y>=90))) needs to be satisfied by the following combinations:

Condition3	Condition4	Condition5
True	True	True
True	True	False
True	False	True
True	False	False
False	True	True
False	True	False

False	False	True
False	False	False

The test cases presented in Table 2-5 satisfy the combination coverage. Note that one of the combinations (Condition 3 is false, Condition 4 is true, and Condition 5 is true) cannot be achieved.

ID	X	Y	C1 (X>=80)	C2 (Y>=80)	C3 (X+Y>=140)	C4 (X>=90)	C5 (Y>=90)	Path
2	95	84	True	True	True	True	False	1-2-3-4-5-6-7-2-8
4	94	90	True	True	True	True	True	1-2-3-4-5-7-2-7-2-3-4-5-7-2-3-4-5-7-2-8
5	70	69	False	False	False	False	False	1-2-8
6	70	90	False	True	True	False	True	1-2-8
7	70	70	False	False	True	False	False	1-2-8
8	90	30	True	False	False	True	False	1-2-8
9	30	90	False	True	False	False	True	1-2-8
10	N/A	N/A	N/A	N/A	False	True	True	N/A

Table 2-5: Test cases that satisfy the condition-combination coverage criterion

All of the coverage criteria described so far can be arranged in a hierarchy of strength from weakest to strongest as follows: statement, decision, decision/condition, combination. The stronger the criterion, the more defects can be revealed by the tests.

2.8.6 Path Coverage Criterion

Path adequacy coverage is stronger than the criteria mentioned above. The test cases that satisfy this criterion require that all paths be exercised, including loops, decisions and conditions. Hence, a large number of test cases need to be designed. This is usually infeasible in practice, as a large number of test cases may be needed. Moreover, it may not be possible to design test cases for all paths. As an example, suppose the following code excerpt:

```
1. If (!A)
2. B++
3. Else B--;
4. If (!A)
5. D--
6. Else D++;
```

Figure 2-19: A code excerpt for illustrating path coverage criterion

This code contains six statements with four possible execution paths: 1-2-4-5, 1-2-4-6, 1-3-4-5, 1-3-4-6. However, only paths 1-2-4-5 and 1-3-4-6 are possible.

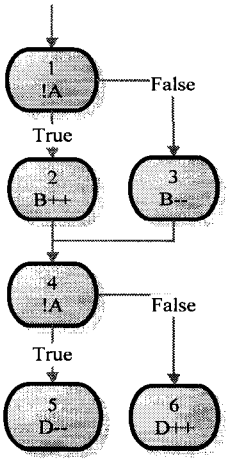


Figure 2-20: A control flow-graph representation of the code excerpt in Figure 2-19

Another problem with path coverage is that a large number of paths may be needed in the presence of loops. For instance, in the code sample in Figure 2-17, there is a "while" loop, that is terminated on condition (X<80 and Y<80). The number of repetitions of this "while" loop depends on the execution of statements within the "while" loop. The number of repetitions can be extremely large, even infinite. A repetition limit may be needed to obtain a reasonable number of paths.

2.9 Where Does Our Approach Lie In the SDLC

Figure 2-21 presents a software development life cycle (SDLC). Use cases are used to represent functional requirements in the requirement-engineering phase. Recall that a test case includes a set of test conditions, test inputs, and expected results, and that it is developed for a particular objective or to validate a specific functionality in the application under test. Test conditions come from use case pre-conditions. Input output sequences (test scenarios) are derived from use case events. In order to be executed, concrete test cases must be produced using data values for test conditions and test scenarios. Each test condition may correspond to a multitude of test cases. Specifications evolve to detail designs normally represented as object diagrams, class diagrams etc. Test cases can guide the detail design. After the software implementation, dedicated testers can perform testing tasks with reference to the generated test cases and identify the defects. Since the requirement specifications are inevitably changing during the software development life cycle, our approach is built in an automated way so as to make the process efficient.

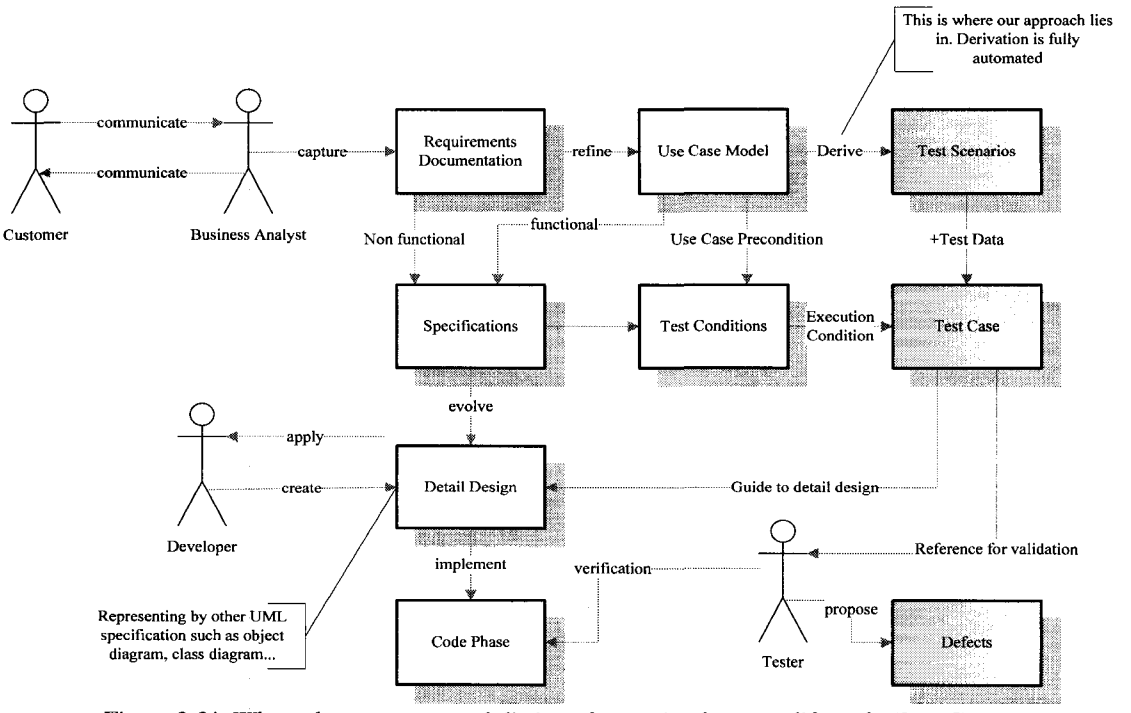


Figure 2-21: Where does our approach lie in software development life cycle (SDLC)

2.10 Chapter Summary and Highlights

Testing is often seen as a final stage of software development. It should instead be one of the first tasks in every project. Use case modeling is an effective technique that tells stakeholders what the system does and how it interacts with actors as well as what systems react when the consequence of failure happens [Coc01]. Use case description is important, as it leads systematical identification of test cases for the system. However, a deficiency is proposed that the three relationships in the use case model are not able to describe the implicit use case dependencies. To keep the flow of events on a system level, these dependencies should be captured.

When generating test cases from use cases, test coverage analysis is required. The degree of coverage makes sense in terms of the type of software, its mission or safety criticalness, and the time and resources available. We introduce six test coverage criteria in the white box-based design approach from the viewpoint of the control-flow graph. Since our test case generation approach can occur in the requirement phase of software development and it is far earlier than normal white box approach can be conducted, we call it a grey box-design approach.

Chapter 3 - LITERATURE REVIEW AND RELATED WORK

3.1 Introduction

Despite continuous refinement of requirement specifications during the software development life cycle (abbr., SDLC), they will never be perfect. Test cases crystallize the vision of a software product's behaviour as specified in the requirements. They are used to validate requirements by revealing lack of clarity, omissions and ambiguities against requirements. As test cases are so beneficial to requirements, having test cases on hand can keep changes to requirements on the right track.

Generating test cases from requirements is not a new idea. Many researchers have proposed a variety of approaches to achieve this goal. The relation between UML specifications and testing is the subject of a number of recent works. In this chapter, some recent research results employing UML artifacts for testing purposes will be summarized. A conclusion about the features of our approach will be proposed in the last section.

3.2 Literature Review of Model-Based Testing

Model-based test-case generation approaches are proposed in many literatures such as [Bri02] [Cav04] [Frö00] [Rys99] [Wan04]. UML artifacts are widely used as the models [Bec02] [Bri02] [Rys99] [Cri01]. Riebisch et al. [Rie05], Pickin et al. [Pic00] and Briand et al. [Bri02] derive test cases from either system sequence diagrams or other UML 2.0 diagrams. Kansomkeat et al. derive test cases by transforming statecharts to test flow graphs [Kan03]. The literature under review can be distinguished into three major categories: state machine-based, scenario-based and state machine & scenario-based. We chose a representative literature for each category to compare with our approach.

3.2.1 State Machine-Based

With respect to UML-based modeling for system-testing purposes, finite state machines (abbr., FSM) are widely used as an intermediate format. However, few literatures discuss the approach of generating test cases from use cases solely by using state machines. In these literatures, Fröhlich et al.

[Frö00] propose a semi-automated method to generate test cases from use cases. In their approach, a use case description in text style is formally transformed into a UML statechart. Test cases are then generated with a given coverage from the statechart by mapping statechart elements to an AI planning formalism: STRIPS[1]. The generated test cases include test sequences and constraints of test data. Table 3-1 presents a comparison table that summarizes the difference between Fröhlich et al.'s approach and our approach. The two approaches are compared from three aspects: system-level, automation-level and test coverage.

- ***System-level***

An entire system can be modeled as a UML use case model and the functions of this system are decomposed by adding UML use case relationships. However, Fröhlich et al's approach only considers include relationships when transforming use cases to statecharts. An include relationship is transformed to statechart elements by creating a sub-machine (formalizing the included use case) and a stub-state (connecting the states in the sub-machine to the right state in the enclosing machine). The test sequences generated from the sub-machines are just simply added into their surrounding machines. Figure 3-1 presents an example of the statechart from [Frö00].

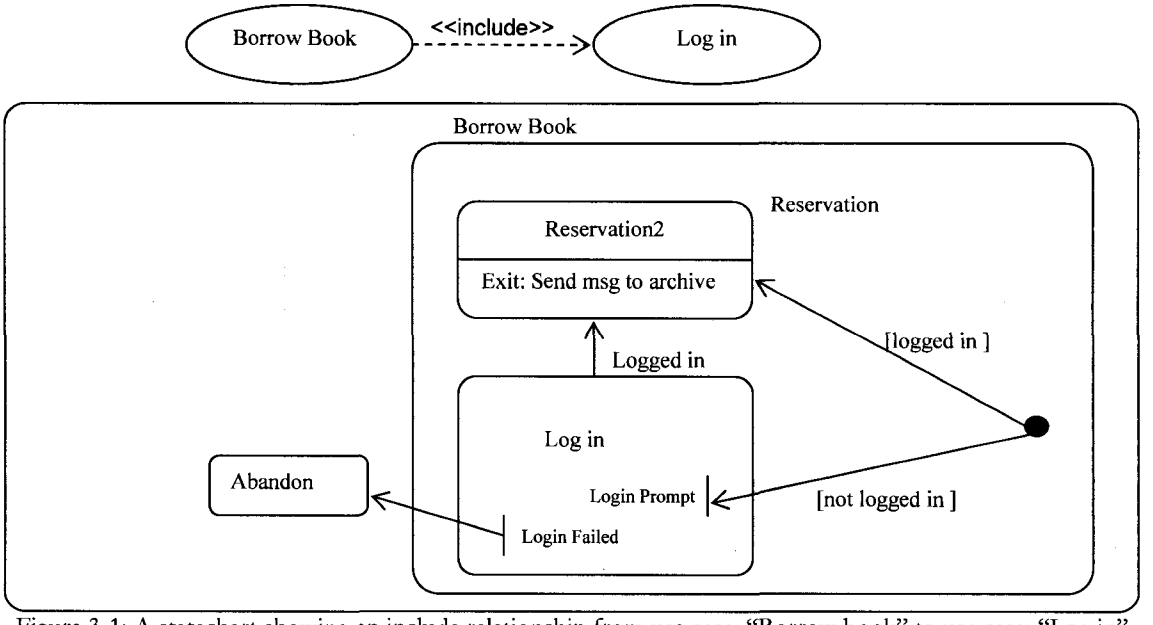


Figure 3-1: A statechart showing an include relationship from use case: "Borrow book" to use case: "Log in" [Frö00]

[1] STRIPS describes states & operators in a restricted language. It is the simplest and the second oldest representation of operators in AI.

The included use case: "Log in" is a sub-machine, the state (e.g., Login Prompt, Login failed, etc.) within the sub-machine are connected to the right state. The statechart does not have notations denoting extend relationships. Moreover, recall that a limitation of UML use case relationship is proposed in Chapter 2. Fröhlich et al's approach does not take into account the sequential relations to avoid too much functional decomposition. In our approach, we transform the extend relationship when mapping use cases to control flow-based state machines (abbr., CFSM). In addition, the sequential relations among different use cases are captured before generating test sequences from the state machines.

- ***Automation-level***

Fröhlich et al's approach employs STRIPS planning to generate test cases. The planning starts from the initial state of a state machine where the execution of the test begins. Then, a set of propositions (each state is described by a set of propositions, which must hold before that state is visited) will keep the track of a program execution. A set of operators describes the transitions among the states. Each operator describes which propositions are established or possibly deleted by the application of the operator. After the planning task finds a sequence of operators that connect from the initial state until the final state, a test case is yielded. The automation of the planning is debatable. Although all propositions can be collected, they do not include system reactions, which means these system reactions must be added into the test case manually. Compared with the Fröhlich et al's approach, our approach captures the propositions (we call them "predicates", which will be introduced in Chapter 5) through an automated tool - Domain Model Editor (will be introduced in Chapter 4). When mapping use cases to CFSMs, the system reactions are considered. Thus, yielded test cases will involve the system reactions. Another limitation of Fröhlich et al's approach is that there are too many detail designs (e.g., collecting propositions and formalizing) of each state that require user involvement. In contrast, our approach requires fewer user involvement. The users are only required to input all natural language use case specifications, avoiding excessive formalism. However, a limitation of both approaches is that concrete test data must be defined manually.

- ***Test coverage***

Fröhlich et al's approach introduces all-transitions coverage and all-states coverage. They indicated that a complication of the test-sequence generation procedure achieving a given coverage arises in

the case of nested state machines. Accordingly, their approach proposes two trade-offs: test the sub-machines separately and combine the test sequences from sub-machines. However, whether the sub-machines can be tested separately cannot be assumed in advance. Compared with Fröhlich et al's approach, our approach provides all-edge test coverage as well as all-node test coverage. As our approach employs flat CFSMs, the complication of nested state machines can be avoided. In addition, the flat CFSMs are created by combining the included use cases, extending use cases or even the use cases that have sequential relations. The generated test cases depend on the combination of above relationships indicated by the user. Moreover, we propose complete-path coverage. Thus, the loops (i.e., the recovery scenario) can be tested.

	System-level	Automation-level	Test Coverage
Fröhlich et al's approach	♦ Test sequence combinations are only limited between sub-machines and surrounding state machines (An include relationship) ♦ Extend relationships and implicit use case sequential relations are not considered.	♦ System reactions must be added into the generated test cases manually. ♦ Many detail designs in regards to the STRIPS are required before transforming use cases to statecharts. ♦ Concrete test data must be defined manually	♦ All-state and all-transition test coverage are proposed. ♦ Complication of the nested state machines from state charts arises.
Our approach	♦ Test sequences from an extension use case denoted by an extend relationship are combined into its extended use case automatically. ♦ The implicit use case sequential relations are captured on top of include and extend relationships before transforming use cases to state machines.	♦ System reactions are generated automatically. ♦ Detail designs are avoided before transforming use cases to state machines. ♦ Concrete test data must be added manually	♦ Three kinds of test coverage are provided. ♦ The complication is avoided due to the flat CFSMs.

Table 3-1: A summarize of the difference between Fröhlich et al's approach and our approach

3.2.2 Scenario-Based

Use cases are one of UML artifacts. Conducting testing on typical scenarios ensures that testing focuses on the typical parts of a system. Briand et al. [Bri02] propose a system-perspective testing which concerns testing an entire system from use cases. The name of the project presented in [Bri02] is TOTEM (Testing Object-orienTEd systems with the unified Modeling language). It investigates ways of supporting test drivers[2] derivation from UML artifacts for all levels of the system. Again, we compare TOTEM approach with our approach from three aspects: system-level, automation-level and test coverage.

[2] Test drivers are automated test case executions to avoid repetitive work.

- ***System-level***

In the TOTEM approach, activity diagrams are used to describe sequential relations between use cases. Extend and include relationships are not taken into account when the sequential relations are described in activity diagrams. Instead, the included use cases will thereafter be merged into the sequence diagrams (to identify the use case scenarios) of their including use cases. Although activity diagrams model the control flows of a program, the distinction between normal and abnormal behaviour, which is one of the benefits of use case analysis is lost. Recall that there are two kinds of scenarios: main success scenarios and alternative scenarios. The alternative scenarios can be further distinguished to recovery scenarios and failure scenarios. A use-case flow can follow either from the main success scenario or from the failure scenario of another use case. Analyzing a system from use cases will maintain the benefit from which both sequential relations from main success scenarios and failure scenarios are considered.

Compared with TOTEM, our approach is concerned with the sequential relations on use cases rather than transforming use cases into other diagrams. The sequential relations and extend, include relationships are captured before transforming use cases to CFSMs. A global combined graph, which results from capturing all the relations, will be introduced in Chapter 5. In addition, the use-case flow from a use case will follow the path from the main success scenarios or failure scenarios from another use case where such sequence is applicable.

- ***Automation-level***

Although the TOTEM approach can capture use-case sequential relations from activity diagrams, it is manual in creating the activity diagrams. In other words, the sequential relations that are described in the activity diagrams involve a lot of users' thought. Compared with the TOTEM approach, our approach offers an automated way to capture these sequential relations. The global combined graph is built automatically. The users would prefer to think about whether to add the sequential relations instead of to think about whether a sequential relation exists.

Regarding test data, they must be added manually in both the TOTEM approach and our approach. However, the TOTEM approach considers parameters in use case sequences derived from the directed graphs corresponding to the activity diagrams that describe the sequential relations between use cases. Once the parameterized use case sequences are instantiated, the actual parameters will be

replaced with symbolic values. Our approach provides a manual way to add the test data. Test data automation, however, falls outside of the scope of this thesis and will be addressed in future research.

After deriving use case sequences, the use case scenarios are identified through sequence diagrams in TOTEM approach. The interactions between different objects are described in the sequence diagrams and alternative scenarios are described in this stage. The sequence diagrams are then re-expressed by a regular expression –Object Constraint Language (OCL). Thereafter, the regular expressions are re-expressed to a sum-of-products[3] form in order to identify the conditions for the path realization and specify operation sequences. Briand et al. mentions that they can use a labeled graph and matrix based algorithm to automate the derivation from sequence diagrams to the regular expressions. However, deriving OCL constraints could be quite complicated and automation is debatable.

Above all, the TOTEM approach employs many UML diagrams to derive the test requirements. It involves many detail designs and the approach is heavy. Our approach is lightweight, on the contrary. Use cases and state machines are the only artifacts that required by our approval.

- ***Test Coverage***

The author utilizes a decision table to represent each use case. The test coverage is based on covering variants that are represented in the decision tables. The variants denoting Boolean value "Yes" and "No" can appear as either conditions or messages (e.g., titleNotExist can be either Yes or No). Test cases should cover all variants at least once. It is a derivation of white-box test coverage – condition combination coverage.

Table 3-2 presents a comparison table that summarizes the difference between our approach and the TOTEM approach.

[3] Each product term represent either a use case scenario or a set of scenarios if iteration symbol are represent. The sum-of-products form is product terms separated by "+"

	System-level	Automation-level	Test Coverage
TOTEM approach	♦ Use case sequential relations are captured by describing them in activity diagrams ♦ Include and extend relationships are considered when generating use case scenarios instead of considering them at the same time of capturing use case sequential relations. ♦ Describing the sequential relations in activity diagrams will lost the normal behaviour and abnormal behaviour, which is a benefit that use case analysis has.	♦ It is manual to create the sequential relations describing in activity diagrams ♦ Many detail design when generating test sequences (OCL, sequence diagrams) ♦ Has test data parameter when generating test scenarios, however, the sample test data are still required to input manually.	♦ Test coverage is based on decision tables.
Our approach	♦ The implicit use case sequential relations are captured on top of include and extend relationships before transforming use cases to state machines. ♦ The implicit use case sequential relations are captured without creating activity diagrams. ♦ The sequential relations are captured, and it will also consider the main success scenario and failure scenario when combining different paths.	♦ Offers an automated way to capture sequential relations without using activity diagrams. ♦ Detail designs are avoided before transforming use cases to state machines. ♦ Concrete test data will be added manually	♦ Three test coverage are provided. ♦ The complication of achieving test cases according to certain test coverage criteria is avoided due to the flat CFSMs. ♦ Currently the approach does not support coverage like condition combination coverage, but it will support when the test data can be added automatically.

Table 3-2: A summarize of the difference between our approach and TOTEM

3.2.3 Both Scenario & State Machine-Based

Very few approaches are both scenario and state machine based. Ryser et al. propose a method for SCENario-based validation and Test of software (SCENT) [Rys99] [Rys00]. In this approach, natural language scenarios are formally converted into statecharts. These statecharts are then annotated with information such as pre-conditions, data and non-functional requirements. Finally, test cases are generated by path traversal of the statecharts. In addition, the SCENT approach introduces dependency charts to capture dependencies among scenarios and generate system-wide test cases. One major difference between SCENT approach and our approach is that the specifications employed by our approach are allied to UML, yet the SCENT approach is not. Again, we compare the SCENT approach with our approach from three aspects: system-level, automation-level and test coverage.

- ***System-level***

Recall that in Chapter 2, we have mentioned that use cases[4] and scenarios[5] are regarded as

[4] In SCENT approach, the use case is a sequence of interactions between an actor and a system triggered by a specific actor, which produces a result for an actor.

[5] In SCENT approach, the scenario is an ordered set of interactions between partners, usually between a system and a set of

synonyms in the SCENT approach [Rys99], yet we emphasize that in our approach, use cases[6] and scenarios are two different terminologies. Use cases are regarded at a higher level and more abstract level from scenarios. In the SCENT approach, the author mentions that in order to test applications completely, dependencies between scenarios have to be considered. Each scenario captures a small part of a system's behaviour (thus being partial). Scenarios can be integrated according to different kinds of dependencies. These dependencies can be represented with dependency charts. Ryser et al. distinguish three major kinds of dependencies: *abstraction dependencies*[7], *temporal dependencies*[8] and *causal dependencies*[9]. Figure 3-2 presents more detail categories of the notations for dependency charts [Rys99] [Rys00], all of which are related to the three major kinds. The rectangles with round corners and circular connectors represent scenarios.

- **General Dependency:** The dashed lines that represent general dependencies among scenarios should be named or annotated with required information where appropriate. A general dependency represents a meaning of "is a type of" or/and "a description of".
- **Data/Resource Dependencies:** We illustrate this dependency by using an example. In Figure 3-2, data/resource dependencies describe that the data/resource in scenario "xyz" must be calculated/satisfied before scenario "uvw" can be executed.
- **Sequence (strict and loose):** Strict sequences, which describe that the second scenario must be preceded by the first scenarios and the first must be followed by the second. This kind of sequence is used for temporal dependencies. Loose sequence, which describes that the first scenario must be executed before the second is run (that is, if the second is to be executed, the first must precede it, but the first may well be performed without the second). This kind of sequence is used for causal dependencies.
- **Alternative:** Alternatives may be annotated with the condition(s) and alternatives that can be taken. Often alternatives are quite obvious in naming of the alternative scenarios; hence, the condition(s) is not mandatory.
- **Iteration:** Iteration may encompass as many scenarios as desired, as long as the scenarios are all in the same sequence. An iteration condition (e.g., 0,1,* notation to denote multiplicity,

actors external to the system.

[6] In our approach, the use case contains many different kinds of scenarios, for instance, main success scenarios, recovery scenarios and failure scenarios, which will provide different result for an actor that triggers one of the scenarios.

[7] Abstraction dependencies are established by hierarchical decomposition. Examples are scenarios arranged in hierarchies, scenarios to cover variants and scenarios composed of sub-scenarios.

[8] Temporal dependencies establish a sequence dependency between scenarios. An example is that the second scenario must follow the first; the first must be followed by the second.

[9] Causal dependencies establish a loose sequence, that is, once scenario A has been executed, scenario B may be executed.

or logical expressions denoted by ">", "<", "=") can be attached to the arrow-line.

- **Concurrency:** Accidental concurrency describes that two scenarios may run in parallel. If concurrency must be enforced or is to be prohibited, the scenarios are connected by using "=" or "≠".
- **Real-time:** Real-time dependencies are indicated by the alarm-clock symbol describing that the time delay must be passed before the scenario can be executed.
- **Abstract:** Abstract scenarios are depicted as foldouts. They are to be self-contained in order to be pluggable.
- **Structuring Constructs:** Scenarios that belong together according to some criterion may be packaged in a box.

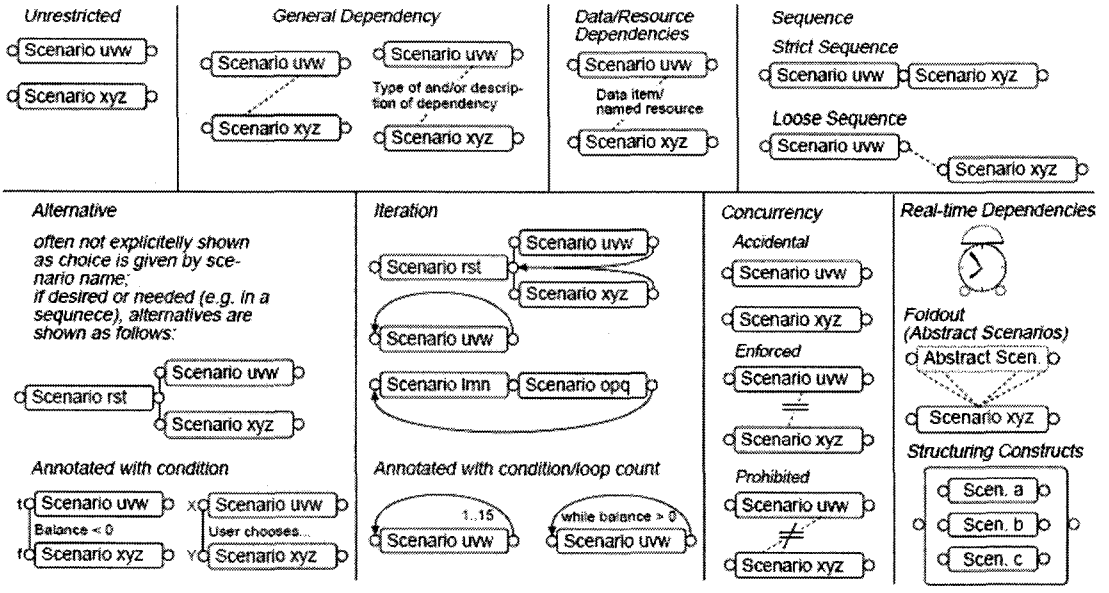


Figure 3-2: The notation for dependency charts [Rys99, Rys00]

Compared with the terminology "dependency" introduced in SCENT approach, our approach proposed a terminology "sequential relations" (see Chapter 2). Our approach is allied to UML use-case specifications, therefore, the sequential relations are considered at a use-case level instead of the scenario level. Although both approaches bear the same objective: to combine the path so that the use-case flow[10] can continue the path in another use case, some advantages of our approach can be found.

- In the SCENT approach, a benefit of use case analysis, which is structuring use cases

[10] In our approach, the use-case flow follows one of the scenarios, which is contained in the current use case.

according to a group of functions, is lost. Therefore, it is difficult to find a group of scenarios corresponding to a use case from the overview diagram[11] presented in the SCENT approach. On the contrary, our approach maintains the benefit of a very clear mapping between a use case and a group of corresponding scenarios.

- In the SCENT approach, it is difficult to choose a group of use cases that users want to generate test cases because they start from scenarios. In addition, we have mentioned that the functional decompositions should be a less consideration. With this in mind, we provide a solution in which users can choose one or any group of use cases (with UML use case relationships, if applicable, or with sequential relations) that they prefer to generate test cases.

♦ ***Automation-level***

In the SCENT approach, Ryser et al. start from scenarios instead of use cases. Despite the fact that the scenario creation procedure[12] is straightforward and easy to apply, the procedure is regarded as having very limited automation. Designing dependency charts from scenarios is very difficult to automate as it includes many detail designs. Moreover, as mentioned in Chapter 2, statecharts are not easy to traverse. Compared with the SCENT approach, our approach brings the following automation:

- **Avoid using dependency charts:** We start from use cases and formalize use cases in Cockburn's format. Although use case specifications and use case models are created manually, our approach can capture the sequential relations among use cases automatically, which avoid using the so-called dependency charts.

[11] In SCENT approach, the overview diagram is an abstract view of the set of all use cases of a system introduced by [Jac92].

[12] Scenario Creation Procedure in SCENT approach: [Rys00]

1. Find all actors (roles played by persons/external systems) interacting with the system
2. Find all (relevant system external) events
3. Determine inputs, results and output of the system
4. Determine system boundaries
5. Create coarse overview scenarios
6. Prioritize scenarios according to importance, assure that the scenarios cover all system functionality
7. Create a step-by-step description of events and actions for each scenario (task level)
8. Create an overview diagram and a dependency chart
9. Have users review and comment on the scenarios and diagrams
10. Extend scenarios by refining the scenario description, break down tasks to single working steps
11. Model alternative flows of actions, specify exceptions and how to react to exceptions
12. Factor out abstract scenarios (sequences of interactions appearing in more than one scenario)
13. Include non-functional (performance) requirements and qualities in scenarios
14. Revise the overview diagram and dependency chart
15. Have users check and validate the scenarios (Formal reviews)

- **Automated use case validation:** The use cases can automatically be validated as being fine before or after capturing the sequential relations, nevertheless in the SCENT approach, the scenarios can only be validated manually.
- **Use case structuring:** Although the SCENT approach introduces the structuring constructs for scenarios, as we have mentioned, even if it is used to build the package, the process is manual. Our approach is at the use-case level, which keeps the functional-related scenarios in a single use case, thus avoiding the manual process.
- **Retain the corresponding dependencies:** In the last section, we introduced different kinds of dependencies represented in dependency charts. In our approach, we have some corresponding sequential relations to these dependencies. For instance, in the dependency charts, there is a notation called real-time. In the use case writing guidelines of our approach, the events "after delay" and "before delay" (introduced in Chapter 2) are used to depict the similar meaning that a period of time should pass before/after the events can continue. In the dependency charts, there are alternatives, iterations. In our approach, the alternatives are described in the use case descriptions and the iterations will appear in the control flow-based state machines in order to involve them in the generated scenario. However, the corresponding sequential relations to the concurrent dependencies and the data/resource dependencies have not yet been explored in our approach. It will be explored in the future research (refer to Chapter 8).
- **Automated state-machine generation:** The SCENT approach uses an unrestricted language in narrative scenarios, thus initially taking a fully informal approach. The scenarios are then formalized in statecharts – a semi-formal representation of scenarios. The approach of transforming scenarios to statecharts however is manual. Furthermore, the mapping of actions and system reactions in natural language scenarios to states or transitions in the statecharts is not definite and clear-cut. Thus, they will be further distinguished manually. Compared with the SCENT approach, our approach creates the CFSMs automatically and they are enhanced to a system level by adding sequential relations (More details can be found in Chapter 4). All elements in a test scenario will be automatically generated without the involvement of users (see Chapter 6).

♦ ***Test Coverage***

In SCENT approach, test cases from system tests are generated by path traversal in the statecharts.

We indicated in Chapter 2 that statecharts are not easy for traversal; moreover, we mentioned the advantage of CFSMs when comparing Fröhlich et al's approach and our approach in the pervious sections. In the SCENT approach, the author mentions that besides the node and link coverage, more elaborate coverage can be chosen. Data annotations enable the domain tests (boundary analysis, exception testing). Performance and timing constraints allow for performance testing. However, there are no automated methods presented to achieve this test coverage.

Compared with the SCENT approach, our approach has explored white-box test coverage, yet more test coverage can be applied after the concurrent relations (corresponding to the concurrent dependencies in SCENT), data/resource relations (corresponding to the data/resource dependencies in SCENT) and non-functional requirements are integrated into the generated scenarios. These procedures are far more difficult to automate, as more constraints will be considered. It is also possible to import some statistical method to deal with the "explosive scenarios". This part will be left in our future research.

Table 3-3 presents a comparison table that summarizes the difference between our approach and SCENT approach.

	System-level	**Automation-level**	**Test Coverage**
SCENT approach	♦ Dependencies that are described in dependency charts are captured on a scenario level. A benefit of use case analysis, which is to structure use cases according to a group of functions is lost ♦ It is hard to choose a group of use cases that users want to generate test cases, because they start from scenarios at first.	♦ It is manual to create dependency charts from scenarios. ♦ The mapping of actions and system reactions in natural language scenarios to states or transitions in statecharts is not definite and clear-cut. Thus, they will be distinguished further manually ♦ The data information is described on the statechart so that the test cases will include the data, however, sample date must be input manually.	♦ Data, pre-condition, qualities and non-functional requirements are included in statechart, they are considered in test coverage. However, it is manual to achieve these kinds of coverage.
Our approach	♦ Sequential relations are captured on use case level. The scenarios are integrated after users have selected the use cases that they prefer. ♦ Our approach keeps the benefit of a very clear mapping between a use case and a group of corresponding scenarios ♦ Users can choose one or any group of use cases that they prefer to generate test cases.	♦ The process of creating state machines is absolutely automatic. ♦ Avoid using dependency charts ♦ Automated use case validation ♦ Simpler use case content delivery ♦ Use case structuring ♦ Retain the corresponding dependencies from dependency charts ♦ Automated state-machine generation ♦ Concrete test data must be added manually. ♦ The actions, system reactions, guard conditions, time delays are clearly distinguished and automatically generated.	♦ Three kinds of test coverage are provided. ♦ It is easy to traverse and get the test coverage, thanks to the flat control-flow state machines.

Table 3-3: A summarize of the difference between our approach and the SCENT

3.2.4 Other Related Literature Review

In the last sections, we reviewed Fröhlich et al.'s approach, Briand et al.'s TOTEM approach and Ryser et al.'s SCENT approach. The three literatures are representative and similar to our research. In this section, other related literatures will be briefly reviewed.

In section 2.3.3, we indicated a limitation in the UML use case relationships that use case models have implicit dependencies besides extend, include, generalization relationships. Use case sequential relations – the implicit dependencies – are important when considering a whole system. In previous sections, the TOTEM approach was shown to capture the sequential relations by using activity diagrams and the SCENT approach was shown to capture the dependencies by using dependency charts. However, both implicit dependencies are created from user's thoughts, which means they are defined by the users directly. This becomes an obstacle to automation. In [Neb03], Nebut et al. explores the issue further; they have proposed a Use Case Transition System (UCTS) to get the mutual dependencies between use cases in terms of pre/post conditions (a kind of "contract"). In their approach, the pre-condition and post-condition of a single use case have been formalized through logical expressions, combining predicates with logical operators. However, the predicates are generated and compared manually. In addition, in the virtual meeting example of [Neb03], each use case has one flow of event only. Due to the manual comparison and generation of these predicates, we do not think that the approach proposed in [Neb03] can be applied to use cases with a complex flow of events (e.g., alternatives, iterations, etc.). That is to say, if one use case includes a group of scenarios, the approach will be very complicated and almost impossible to realize. Compared with this Nebut et al.'s approach, our approach realized a fully automated way and can be applied to more complicated use cases.

In addition to the literatures that have been reviewed, some literatures combine the technique from other standards or tools, such as XML, XMI, JUnit, Rational Rose, AML, etc. For instance, Riebish et al. employ scenarios and use cases as the basis for test case generation [Rie03]. They transform state diagrams into usage models. A tool named UsageTester exchanges the model data with other development tools via XML (eXtended Modeling Language). Wittevrongel et al. introduce a SCENTOR (SCENario-based testing of Object-oRiented software applications) approach which uses JUnit as a basis for test case derivation [Wit01a] [Wit01b]. The inspiration for SCENTOR

comes from Extreme Programming. Extreme Programming advocates a lightweight development process. SCENTOR aims to support the continual scenario testing which is a prevalent principle in Extreme Programming. The author uses a CASE tool such as Rational Rose to model the scenarios by using UML sequence diagrams, which serves to make the scenarios slightly more formal. This slight formalization partially automates the tedious task of writing automated test drivers. By adding concrete parameters and expected results for each step of the scenario, the test drivers can be easily and logically formed. The UML sequence diagrams are then exported into an XMI (XML Metadata Interchange) format. The XMI file is loaded into SCENTOR with some test specifications and written in XML format. To compare the SCENTOR approach and the TOTEM approach, TOTEM uses much greater UML than SCENTOR in the development process while SCENTOR uses lightweight UML modeling. Users are required to input the concrete parameters, expected results in SCENTOR by using Java instead of OCL, therefore preserving the focus on source code, and avoiding the requirement of developers to learn a new language.

Crichton et al. presents architecture for model-based verification and testing using a profile of the UML [Cri01] [Cav04]. The architecture is a subset of AGEDIS (Automated Generation and Execution of test suites for DIstributed component-based Software) architecture. AGEDIS generates and executes test cases for applications, which are modeled according to the AGEDIS Modeling Language (AML). AML is a specialized UML profile and describes suitable semantics and addresses the problems of complexity. The model language comes from the Intermediate Format (IF), which is derived from the UML class, object and state diagram.

Despite the fact that many new ideas are constantly generated, the above literatures rely on many other tools, which makes their approaches redundant and give them limited usage. Our approach maintains a simpler way applicable in almost all use case-based requirement engineering phase.

3.3 A Limitation of Generated Test Cases

Recall that all above reviewed literatures impose a path traversal technique to generate test cases, including our approach. However, by applying this technique, a limitation is found that only valid sequences of events can be generated. It is also important to include sequences that are not admissible in the test. For instance, in the use case: "Place Order", the event "Customer selects

items" should appear before the event "System identifies items". A scenario is inadmissible if the event "System identifies items" appears before the event "Customer selects items". Phalp et al. propose a use case enactable model in [Kan05a] [Kan05b] [Pha03]. They also propose intra-use case dependencies and inter-use case dependencies based on the enactable model. The enactable model is used to verify use case descriptions to determine whether the sequence of events in the use case is correct. The approach provides a mechanism for revision, with an opportunity to correct any flaws. The idea of the adjustment of the event for each use case is based on finding a pair of same post- and pre-conditions. The event with the pre-condition should be executed after the one with the corresponding post-condition which has the same value as the pre-condition. An idea of reaching these inadmissible scenarios is to record the sequences of events which are flawed. Though the inadmissible scenarios are not within the scope of our thesis, Phalp et al.'s intra-use case dependencies inspire us that in the future, users will not be required to write the use case descriptions according to the normal order. An automated mechanism will check each event and propose the admissible sequences and inadmissible sequences. In addition, the number of scenarios will increase due to the combinatorial explosion; some coverage is required when this process is automated.

3.4 Chapter Summary and Highlights

State machine-based approaches are usually accurate but require much more expertise and effort. It is a major undertaking for the development process. Scenario-based testing approaches are usually more easily applied, but provide only partial results. The approach presented in the thesis shares the state machines and scenario based testing. However, a key difference between our approach and the above approaches is that we propose automated generation of test scenarios from the original use cases in a slightly formalized text form such as Cockburn's use case format. In [Frö00] [Rys99] [Rys00] state machines are first derived from use cases but in [Bri02], scenarios represented as UML interaction diagrams need to be derived from the use cases. The derivation of state machines or interaction diagrams from use cases involves some design decision. Our objective is to derive system-level tests as much as possible from requirement models. In fact, a primary goal is to test state machines derived from use cases.

Based on the literature review, we propose the following feature of our approach.

- We derive test cases from use cases only, rather than other semi-formalized requirements.
- We avoid using complicated design information during test generation processes in order to simplify the process.
- We try to use a flattened model directly, rather than a statechart which must evolve into a flattened model.
- We try to completely automate the process during the test case generation process in order to simplify the stakeholders' effort.

Chapter 4 – USE CASE EDITOR (UCEd) FRAMEWORK AND LIMITATION

4.1 High Level Overview of UCEd

Requirements are obtained from business analysts through communications with clients. Engineers then proceed to detail designs of software. Use case modeling is a way to formalize the requirement specifications for system analysis. Thus, the gap between clients and analysis can be filled. However, the requirement engineering is carried out using manual techniques. Although business analysts contribute a great deal of effort to create clear use case models manually, the results are not satisfactory. An automatic technique is therefore required to minimize the gap for elicitation of requirements. Use Case Editor (abbr., UCEd) proposes such a technique.

UCEd is a tool providing automatic support for requirement engineering. It is a successor of REST (Requirements Engineering with Scenario Tool) [Som06]. The approach in UCEd is rooted in UML. UML [OMG03] is a model driven, extensible language used for modeling system requirements. It is standardized by unifying the best features of different modeling languages. Hence, UML has become an industry standard. UCEd is a hybrid of a model-driven approach that consists of use case modeling, domain modeling and statechart. It supports automated requirement engineering, scenario modeling and scenario simulation to support requirement validation. Selecting UCEd for our work prevents re-inventing the wheel. Our approach, however, can be integrated to frameworks other than UCEd.

4.2 Activities Supported by UCEd

UCEd is a tool for elicitation, formulation, composition and simulation of use cases. The tool takes a set of related use cases written in a restricted form of natural language. It can generate execution specifications such as state models, which integrate the behaviours specified by the use cases [Som05d]. Figure 4-1 describes activities supported by UCEd. These activities compose a use case-based requirement engineering process. The process begins with writing use cases to capture requirements, then a rough domain model is derived from these use cases. It then produces high-level state model specifications of a system as well as clarified use cases and domain models

[Som03]. The generated state models are used as prototypes for the requirements validation by simulation [Som04b]. Scenarios are used to automate the simulation process [Som05a]. In current UCEd release, scenarios can be manually composed from simulation results. By applying our approach, which we will present in the following chapters, the scenarios can be generated automatically (see Figure 4-1). These scenarios can be further used to generate test cases according to certain test coverage criteria.

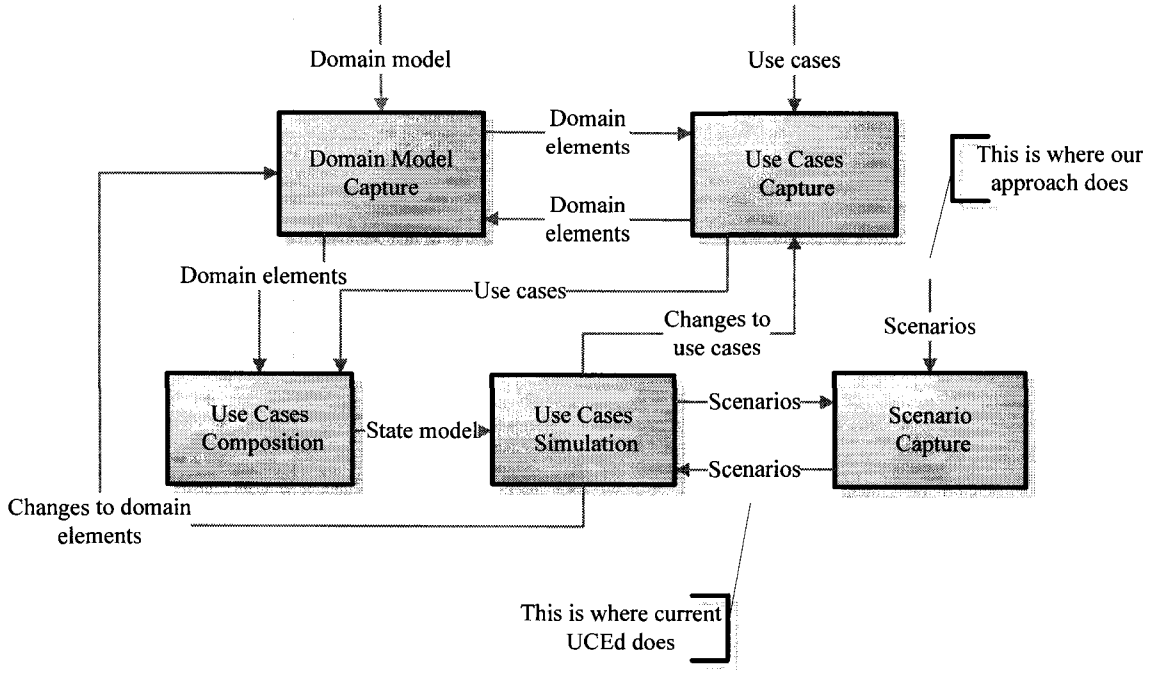


Figure 4-1: Activities supported by UCEd

4.3 Use Case Model

In UCEd, use case capture is done through a separate use case-editing module. The use case-editing module provides an interface that supports editing use case models and use case descriptions. Figure 4-2 provides a view of the interface. It is divided into a left panel and a right panel. The left panel (i.e., use case model-editing panel) is an overview of the Order Process System (abbr., OPSystem) use case model described in Figure 2-1. The right panel is a field-oriented editor (i.e., use case description-editing panel) with each field corresponding to an element of Cockburn's use-case format. It shows the use case description of use case: "Credit Card Payment". Such an editor provides users with an advantage that use cases can be easily created without worrying about delimiting the different fields in use case descriptions. Some key words are underlined in different

colors for easy identification so that the user will not get lost in complicated use case descriptions. After the use case model and its detailed descriptions are composed, UCEd can check use cases and domain models against each other. Inconsistencies and omissions will be reported.

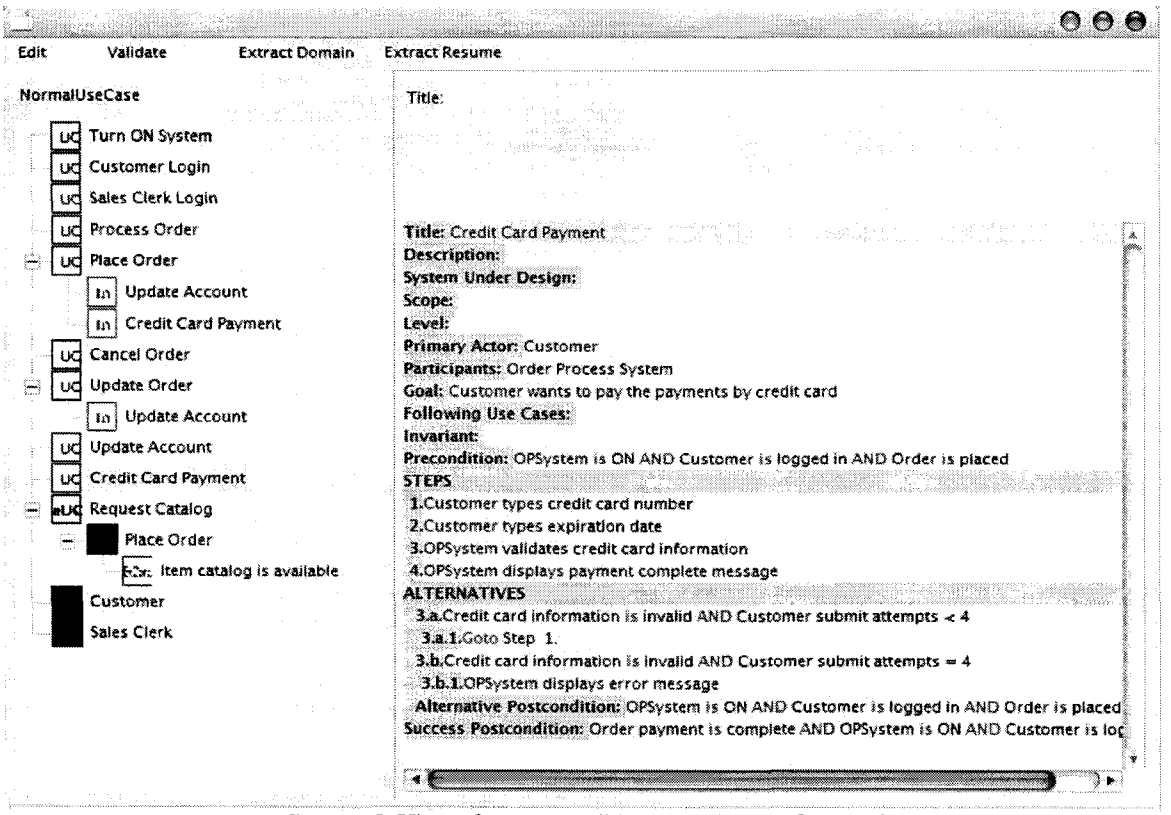


Figure 4-2: View of use case-editing module of UCEd tool

According to Cockburn's use-case format and our use-case writing guidelines, the use case pre-condition must be satisfied prior to the execution of the use case. The use case: "Credit Card Payment" in Figure 4-2 has four normal steps. Normal step *3* has two alternatives *3.a* and *3.b.*, which are executed under specific alternative conditions. Alternative use case step *3.a.1* is a branching statement, which passes the control back to use case normal step *1* and will continue the control flow thereafter. The remaining use case steps of this use case are instances of concept operations. There are two kinds of post-conditions. The condition evaluated at the last use case normal step of the main success scenario is called a success post-condition. The condition evaluated at the last use case alternative step of the failure scenario is called an alternative post-condition. No post-conditions are required for alternative steps with recovery scenarios.

4.3.1 How does UCEd Models Use Case Sequential Relations

A use case diagram depicts use case names, actors, relationships between actors and use cases, and relations between use cases. Besides include and extend relationships (UCEd does not support generalization relationships currently), UCEd provides a "precede" relation. The precede relation explicitly states that a use case is followed by another use case. It is used to integrate the behaviours defined by separate use cases [Som05d]. Users can add precede relations in the use case model-editing panel as well as the corresponding "resume operations" in the use case description-editing panel. The precede relation is not a part of UML specifications. It is a relation created only in UCEd to enhance use case models to a system level. Figure 4-3 presents the system-enhanced OPSystem use case model. Three precede relations are added in the use case model-editing panel and their corresponding resume operations are added in normal step *8* in the use case description-editing panel.

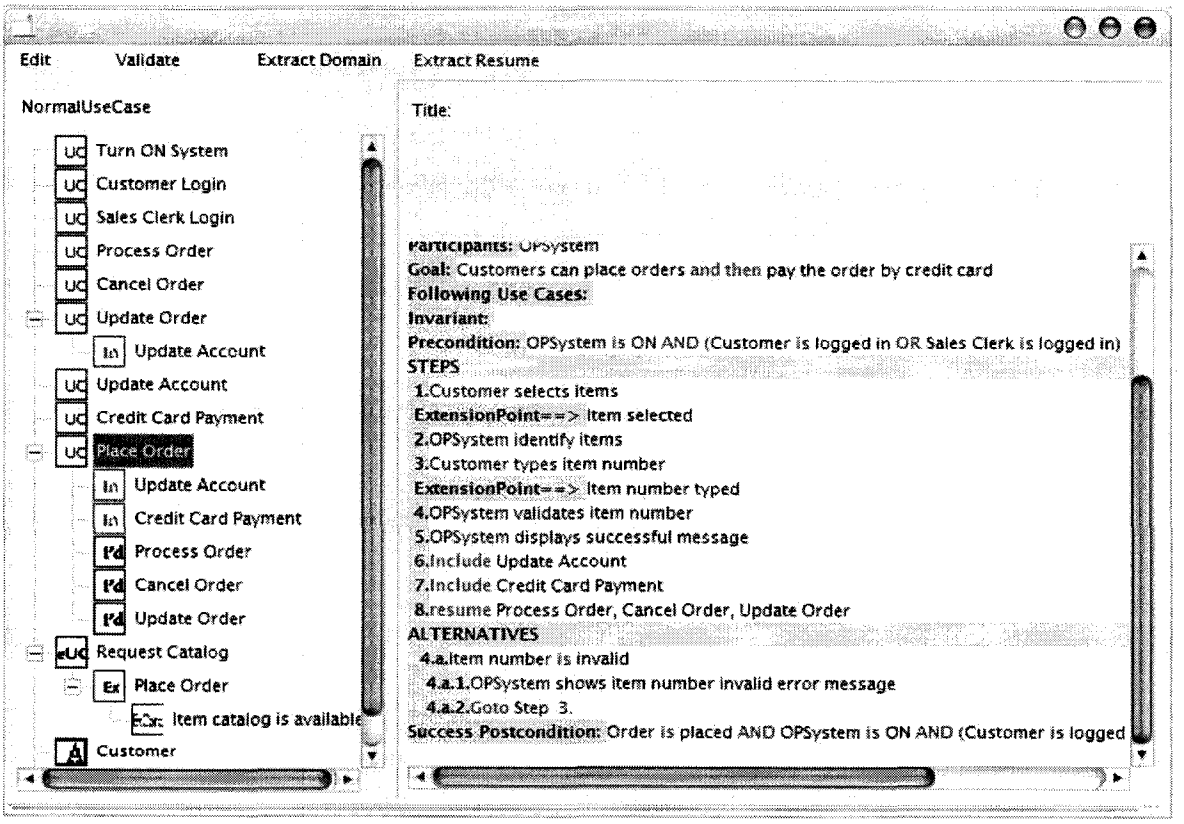


Figure 4-3: Enhanced use case model and use case description from Figure 4-2

4.4 Domain Model

It is important to have a domain model for use case modeling, because domain model can facilitate validation of use case descriptions. A domain model can be further refined to a dynamic object model, which creates a web of interconnected objects, where each object represent a meaningful individual. Domain models are always done after drawing use case models because entities can be extracted from the use cases and put into the domain model as concepts or system concepts.

A domain model is a high-level class model. In UCEd, domain elements capture can be done through either a domain model editing module (Figure 4-4, left) or an automated domain extraction tool (Figure 4-4, right). In addition, a validation mechanism is provided to ensure that an adequate number of domain elements have been captured. Capturing domain elements allows use-case syntactical analysis as well as state model generations from use cases and domain models.

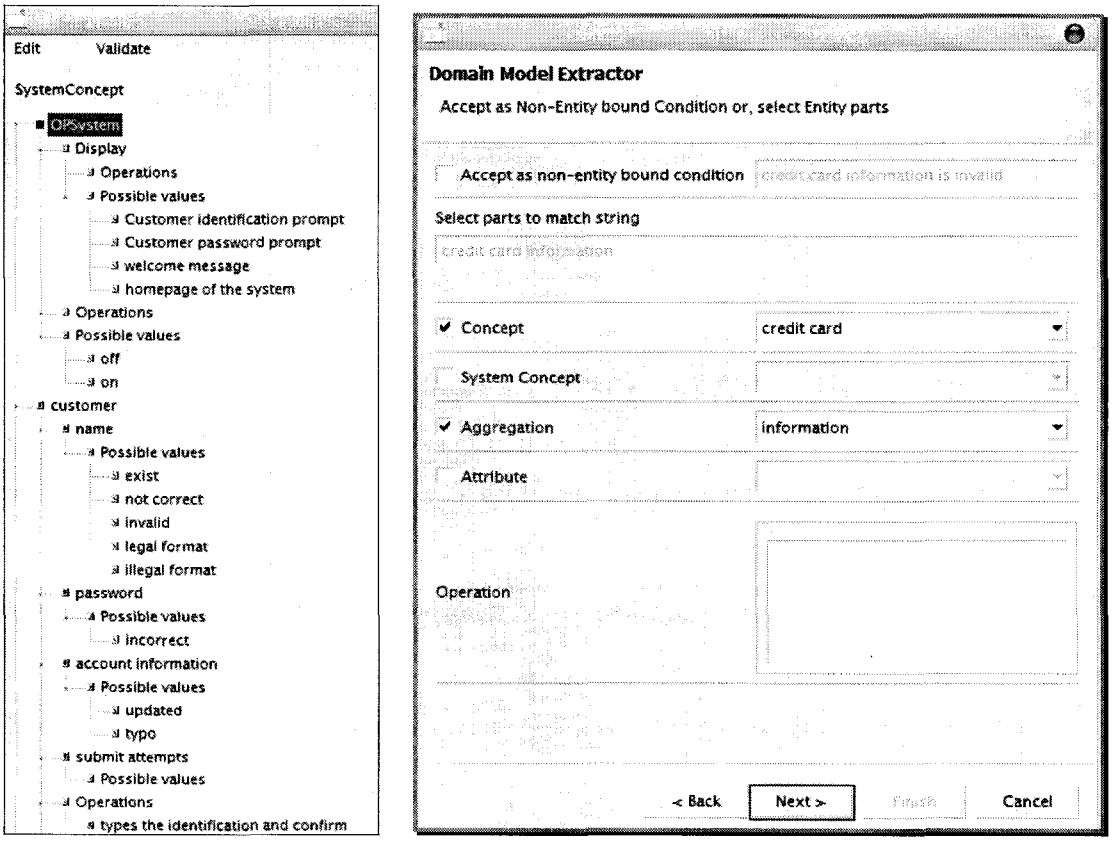


Figure 4-4: Views of domain model editor and domain model extraction tool

4.5 State Model

UCEd employs state models as frameworks for use case integrations. As use cases are created incrementally, state models are formed by merging partial behaviours of each use case. State model specifications contain all partial behaviours of use cases. During this process, any inconsistency found with the domain and use case models will be detected and reported.

UCEd provides two approaches of generating state models: a state model-synthesis approach that is based on control flow and a state model-synthesis approach that is based on operation effects. The control flow-based generation is appropriate at earlier stages when operations have not been specified. The synthesis based on operation effects is useful to validate contract specification of operations and is therefore more appropriate in later stages [Som05d].

Control flow-based generation generates a control flow-based state machine (abbr., CFSM), which is a kind of finite state machine (abbr., FSM). The generation is based on a predefined algorithm. We state below the basis of the functionality of the algorithm.

- Generate a control-flow graph from a single use case. The main success scenario and alternative scenarios are modeled in one CFSM. States and transitions are labeled in a fashion that helps correlate them with corresponding entries in respective use case descriptions.
- Augment CFSM considering extension use cases and inclusion use cases. The extensions use cases and inclusion use cases are incorporated by adding more states and transitions.

The state models generated in UCEd can be described by using UML statecharts. UCEd generates a hierarchical type of control flow-based statecharts. The statecharts provide more accurate information but compact visual formalization charts. They represent behaviour model descriptions where transitions represent complete interactions. Each transition in the statecharts consists of a trigger (an operation performed by an actor) and a system reaction resulting from the event. Figure 4-5 shows a control flow-based statechart generated by UCEd. It is extracted from the use case: "Place Order" with an extension use case: "Credit Card Payment". A tip message with transition information is shown when we move mouse over state *s8_0*.

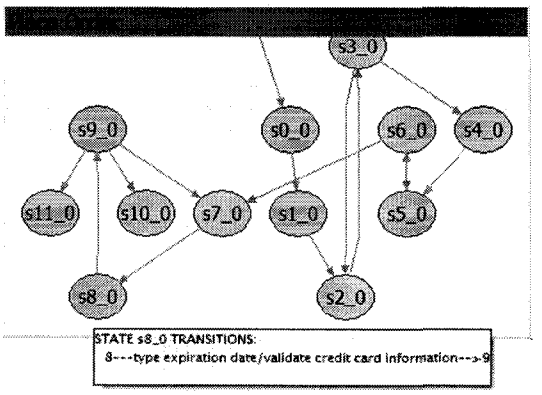


Figure 4-5: Generated control flow-based statechart. In order to avoid cluttering the graph, transitions details can be achieved by moving the mouse on the state or double click the state.

4.5.1 How does UCEd State Model Captures Use Case Sequential Relations

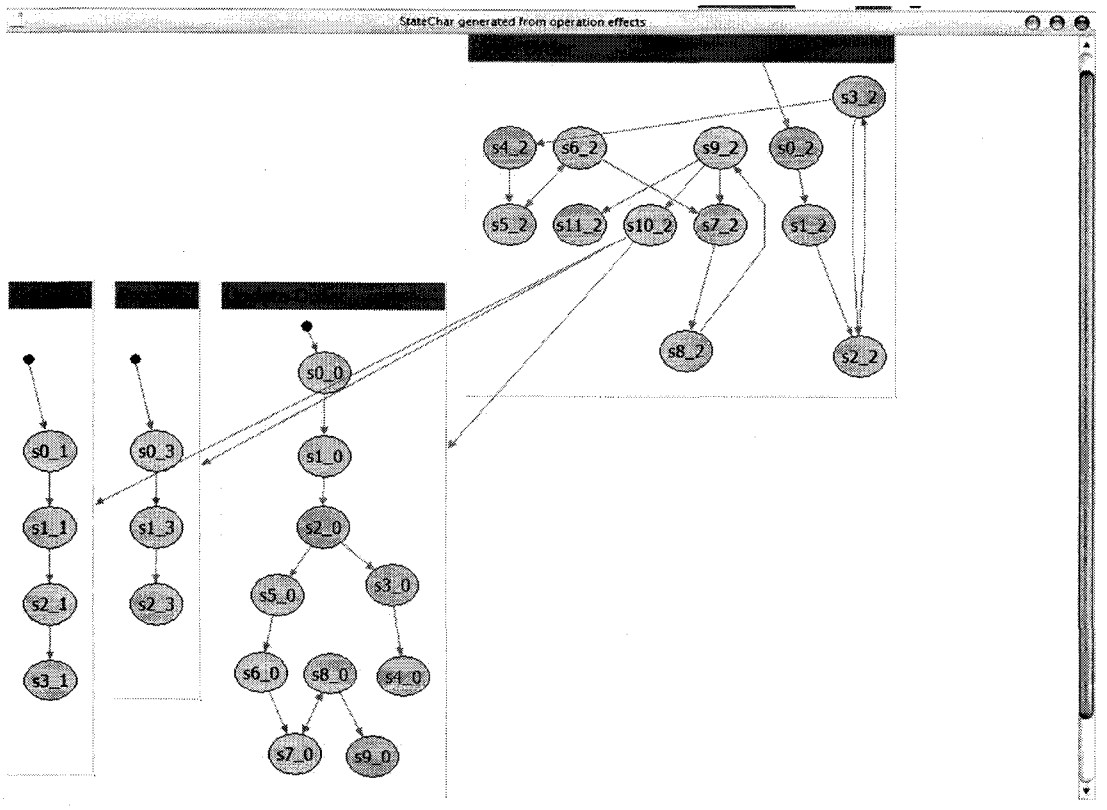


Figure 4-6: Enhanced State Model

In section 4.3.1, we present a system-enhanced use case model by adding precede relations and resume operations. By applying the enhancement, the UCEd state model can be enhanced as well. Figure 4-6 shows a control flow-based statechart generated by UCEd. The generated statechart includes a state for each use case. These states include sub-states and transitions representing use case interactions. A transition to a use case state corresponds to a resume operation (i.e., a precede relation). Such a transition results in the system entering the use case initial state. There are three transitions to the use case: "Cancel Order", the use case: "Process Order" and the use case: "Update Order" from the use case: "Place Order". These transitions are resume operations, which means after the last sub-state *s10_2* in the use case: "Place Order" is exercised, the initial state of the use case: "Cancel Order", the use case: "Process Order" or the use case: "Update Order" can be reached. This is a system-level enhancement on the state model.

4.6 Simulator

A simulation is an effective technique used for requirements elicitation, requirements validation and requirements completion. This technique is able to find the missing and undiscovered requirements of a system. In order to simulate a software system, a prototype needs to be developed. Deriving the prototype manually from requirements can be error-prone and costly.

State machines have the property of being executable and used as prototypes. A state machine generated from a use case includes all use-case flow of events, which constitutes use case scenarios. It can be used as a prototype to validate original use cases and uncover all the scenarios in the use cases. In addition, state machines generated from a use case model can be applied to uncover any possible interactions between use cases.

In UCEd, simulations are conducted through a graphical user interface. The simulator tool allows use case simulations by using generated statecharts as prototypes. By applying this simulator, we can reproduce the reactive behaviour described in use cases and exhibit the global behaviour resulting from their integration. Figure 4-7 shows a view of UCEd simulator tool for the OPSystem. The simulator includes an actor events panel (left) and a simulation panel (right).

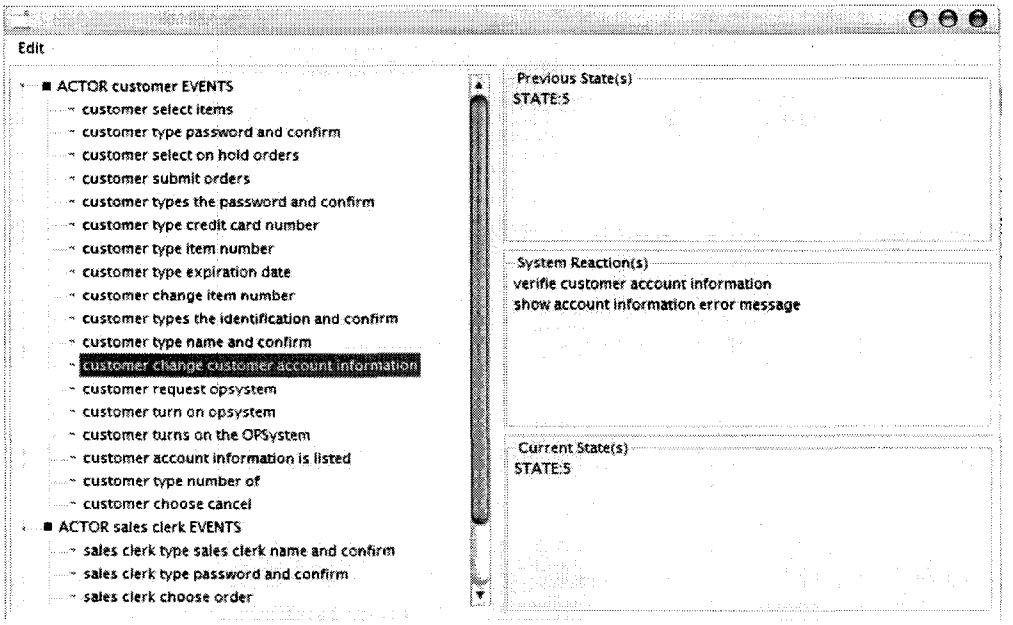


Figure 4-7: View of simulator tool

4.7 Scenario Model

A scenario describes a sequence of interactions between a system and actors. The sequence is composed of a flow of events. A scenario can either be limited to a single use case or used across several use cases. UCEd provides a scenario-editing tool. Figure 4-8 is a view of UCEd scenario-editing tool.

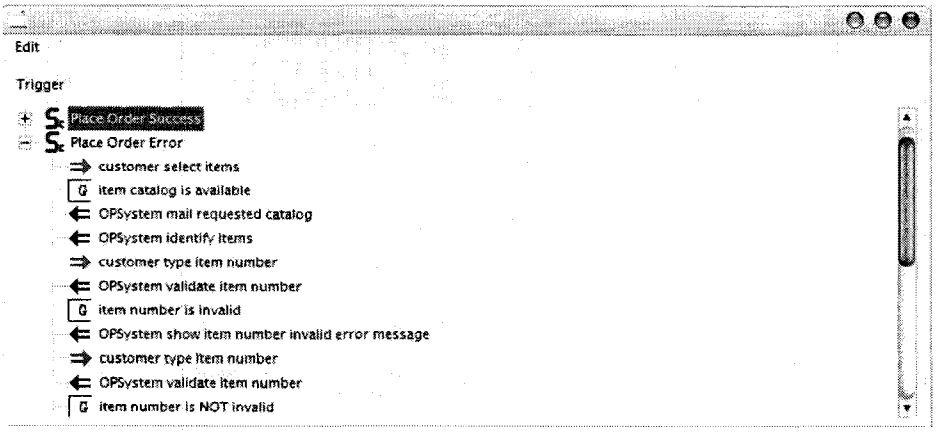


Figure 4-8: View of scenario-editing tool

Scenarios are very useful to document interactions of interest that may or may not be desired and to serve as repeatable scripts for simulation. As discussed in Chapter 2, a scenario is a sequence of:

triggers, system reactions, waiting delays, guard conditions and assertions [Som05d].

- A *trigger* is an operation of an actor of a system.
  Line *1* with the arrow pointing to the right is a trigger. "Customer selects items" is a trigger corresponding to operation "select item" of concept "customer".
- A *system reaction* is an operation of the system under consideration.
  Line *3* with the arrow pointing to the left is a system reaction.
- A *waiting delay* specifies a point in a scenario where a certain amount of time passes without any trigger or system reaction. Timeouts may be enabled by waiting delays during scenario execution.
- A *guard condition* is a condition set to hold at a certain point in a scenario.
  Line 7 with a "G" sign is a guard condition.
- An *assertion* is a condition that needs to be true at a certain point in a scenario.

4.8 Limitations of UCEd Framework

UCEd provides a process for use case-based requirement elicitation. This process is composed of the modules we have just proposed. These modules can be completed through either automated or manual means. One limitation that has already been improved is an automated domain extraction tool. Before that, users could only capture domain elements manually, which required many efforts on manual validation.

In the current UCEd release, users can only write use cases manually. Use case sequential relations can only be captured and represented through adding a resume operation manually at the end of a use case description as well as a corresponding precede relation. It is tedious and time-consuming work if a system is described by a great many use cases. In addition, it is very difficult for users to determine where they should add resume operations and precede relations. An automatic extraction of these resume operations as well as precede relations is more effective and efficient than cumbersome and manual ways.

Another limitation is that scenarios are simulated by users manually. UCEd does not have an automated way to create scenarios. The users must create scenarios based on their simulation results or directly from use cases. An automated way of generating test scenarios based on some test

coverage is very efficient to create further test cases.

Our final objective for UCEd is to generate concrete test cases automatically. Therefore, we will present some extension tools based on a fully automated method of generating system-level test scenarios. These tools will build an efficient way to the test case generation.

4.9 Chapter Summary and Highlights

In this chapter, we presented the UCEd framework by introducing different modules of UCEd. These modules compose a requirement engineering process as described in Figure 4-1. UCEd employs use cases for requirement elicitation and domain models for use case validation. The scenario models are for requirement simulation and they can be used for testing based on some test coverage. UCEd use case models and state models can be enhanced to a system level. The enhancement process however is performed manually, which is very tedious. Moreover, the way of creating scenarios in the scenario models is also manual. There are two limitations in the current UCEd. We will present our approach to solve the two limitations in the following chapters.

Chapter 5 – INFERENCE OF USE CASE SEQUENTIAL RELATIONS

5.1 Introduction

System testing concerns testing an entire system based on its specifications. System testing can ensure functional compliance of an application with its requirements. Use cases that are used as functional requirements specifications can be used as a basis for test derivation. For instance, in section 3.2.1, we reviewed a semi-automated test-case generation approach from use cases by Fröhlich et al [Frö00]. In this approach, use case descriptions consisting of pre-conditions, post-conditions and scenarios written in natural language are used to generate system-level test cases. However, as discussed in section 2.3.3, UML use case relationships are not sufficient to capture all sequential relations between use cases. These sequential relations cannot be omitted when test cases are generated at a system level. The flow of events in one use case may only be activated following a specific flow of events in another use case.

In this chapter and the next one, we will present an approach of test case generation from use cases. Implicit sequential relations between use cases are analyzed based on pre-conditions and post-conditions. Both success post-conditions and alternative post-conditions are considered. These conditions are represented as predicates. Subsequently, we propose the effect of generalization relationships on use case dependencies. Thereafter, implicit sequential relations are explicitly described in a graph. They are at the same level as UML use case relationships in use case diagrams. Since the sequential relations are not allied to UML specifications, we rename the use case diagram with sequential relations a "global combined graph". Scenarios are generated from global control flow-based state machines (abbr., CFSM), which are composed of several CFSMs with the sequential relations. Test scenarios generation is in light of certain test coverage so that the test cases will be more practical. The approach is implemented and integrated as a vital part in UCEd – a tool for use case-based requirement engineering.

5.2 Use Case Condition Representation using Predicates

Generally, a predicate is an operator or a function that returns a Boolean value "True" or "False". Recall that in Chapter 2, the predicate X>80 is an operator. For instance, it returns a Boolean value

"True" when X is equal to 81 and returns a Boolean value "False" when X equals to 79. The predicate X>80 is said to hold when it returns a "True" value. In our use-case writing guidelines, use case conditions are written in a natural-language form where grammar cannot be escaped. When referring to the natural-language grammar, a predicate is one of the two constituents of a sentence. The predicate is the entire sentence except for the subject. For instance, in the sentence "Order Process System displays an error message", the "displays an error message" is a predicate, and the subject is "Order Process System". Technically, the simple noun "system" is the subject and the verb is "display"; the other words modify the key terms and comprise elements of the subject – the subjective phrases and the predicate phrase.

A use case condition is such a sentence that consists of subjective phrases, verbs and predicate phrases. In the use-case abstract syntax presented in Chapter 2, we introduced that use case conditions include pre-conditions, post-conditions, alternative conditions, extension conditions and operation conditions. A use case condition may consist of several atomic conditions. Each atomic condition corresponds to a predicate. A non-atomic condition corresponds to a compound-predicate, which are constructed using Boolean operators.

In our approach, a predicate is used to denote a logical condition in a formal way. To formalize use case conditions by using predicates, the predicate can be denoted as a pair $<E, V>$, where E is an entity and V a value [Som05b]. Entities refer to subjects or subjective phrases. In use cases, the subjects can be either the actors or the system under consideration. Values are distinguished into atomic values and set values. The atomic values are denoted as units such as "ON" or "logged in", while set values are denoted by way of comparisons such as "less than 4 (< 4)". An entity can either be atomic or set. An atomic entity can only be used in the predicates with atomic values, while a set entity can only be used in the predicates with set values. A condition corresponding to a predicate $p = <E, V>$ is said to hold (at an evaluation moment t), if p evaluates to "True" (at moment t) (i.e., a valuation of E is V at moment t). For instance, the condition "Customer is logged in" is formally a predicate <Customer, logged in> that is said to hold, if the entity "Customer" has a value "logged in" at the evaluation moment. An example of the set value "less than 4" is in the condition "Customer submits attempts are less than 4". The entity "Customer" is of type atomic, while the entity "Customer submits attempts" is of type set. When denoting non-atomic conditions, we use

the classical Boolean logic operators, conjunction (AND), disjunction (OR) and negative (NOT)[13]. As an example, the pre-condition for the use case: "Place Order" is "Order System is ON AND "Customer is logged in OR Sales Clerk is logged in". It involves a conjunction operator and a disjunction operator.

5.3 Use Case Dependency Analysis

"include", "extend" and "generalization" are three type of use case relationships defined in UML. It is possible to know the direction of the execution paths if use case dependencies are described by the three relationships. However, these execution paths are not sufficient to generate test scenarios. As discussed in Chapter 2, the three UML use case relationships do not capture use case sequential relations. Another limitation is that the rule of execution path in the case of generalization is not well defined. It will create flaws when generating test scenarios. In this section, we first clarify the effect of the generalization relationship on use case dependency analysis. Thereafter, we present an approach to infer the sequential relations based on use case pre-conditions and post-conditions. By applying this approach, the execution path will be expanded with the sequential relations. In subsequent sections, we present the implementation of this approach in UCEd.

5.3.1 Effect of UML Use Case Generalization Relationship on Dependency Analysis

Recall that in Chapter 2, we have introduced the generalization relationship with control flows. However, there are different situations for generalization relationships that require different solutions. For instance, Figure 5-1 lists different kinds of generalization relationships. To the left is a parent use case specialized by one child use case. In this case, to consider the use case dependencies from a control-flow view, an execution path will start from the child use case UC_2 after it is instantiated. Then, the execution path will follow the basic flow of the parent use case UC_1. When the execution path reaches an abstract point, it will follow the corresponding concrete flows in the child use case UC_2. Therefore, the parent use case is also called an abstract use case, and the child use case is called a concrete use case. However, as we know from our knowledge, a parent use case may have two or more child use cases (Figure 5-2, the one in the center and the one to the right). In

[13] In the three Boolean operators, the priority is NOT>AND>OR if there are no brackets to separate them.

addition, the parent use case may have more than one abstract point, which means the execution path may direct to different concrete flows in different child use cases. Thus, confusing problems will arise:

- *How are child use cases in different abstract points chosen?*
- *How will the execution path direct to combine the scenarios?*

To solve the problem, we categorize the solution presentation into three types:

- **Type 1:** One child use case and more than one abstract point in its parent use case.
- **Type 2:** More than one child use case and only one abstract point in their parent use case.
- **Type 3:** More than one child use case and more than one abstract point in their parent use case.

The "more than one abstract point" refers to the abstract point with the same content. For instance, the abstract point "Payment Method" will lead to concrete use cases that relate to "Payment Method". "More than one abstract point" means that "Payment Method" will appear more than once in the course of a parent use case. It is possible that the abstract point "Payment Method" and another hypothetical abstract point "Order Package Method" will appear in the course of a parent use case together. However, this is not the situation under discussion.

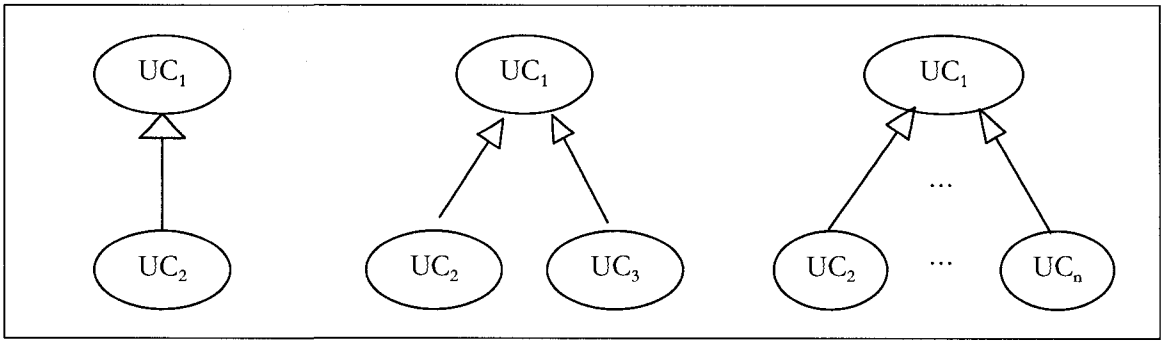


Figure 5-1: Generalization with one child use case, two child use cases and n child use cases

5.3.1.1 One child use case and more than one abstract point in its parent use case

In the case of a parent use case with one specialized child use case, if there is more than one abstract point in the parent use case, every time the abstract point is reached, the execution path will follow the concrete flow of events in the child use case. In Figure 5-2, to the right is an example, fragments of use case descriptions, which corresponds to the general description on the left. The use case: "Arrange Payment" has an abstract point "Payment Method". Due to its alternative course,

the abstract point will be reached more than once (actually twice). Once the execution path reaches the abstract point "Payment Method", it will follow the concrete course of the child use case: "Credit Card Payment". Figure 5-3 is an illustration of the execution path. The black round dot "●" means that the use case is instantiated. The black square "◆" indicates the abstract point.

[Parent use case (abstract use case UC_1)]
Basic Course:
1. Step 1
2. *abstract point*
3. Step 3
4. *abstract point*

Alternative course:
...

Sub-course:
{abstract} abstract point

[Child use case (concrete use case UC_2)]
Basic course
...
Sub-course: *abstract point*
1. Step 1
2. Step 2
3. Step 3

...

Example:
[Parent use case: "Arrange Payment"]
Basic course:
1. System displays Payment method selection
2. Customer selects *Payment Method*.
3. System displays order number and record.
4. System displays Payment complete

Alternative course:
1a. After 60 seconds
 1a1. System alarm warning
 1a2. GOTO 1

Sub-course:
{abstract} Payment Method

[Child use case: "Credit Card Payment"]
Basic course:
...
Sub-course: *Payment Method*
1. Customer selects Credit Card Payment
2. Customer types credit card number
3. Customer types expiration date
4. System validates credit card information

Figure 5-2: An example showing one child use case specialized from its parent use case with more than one abstract point

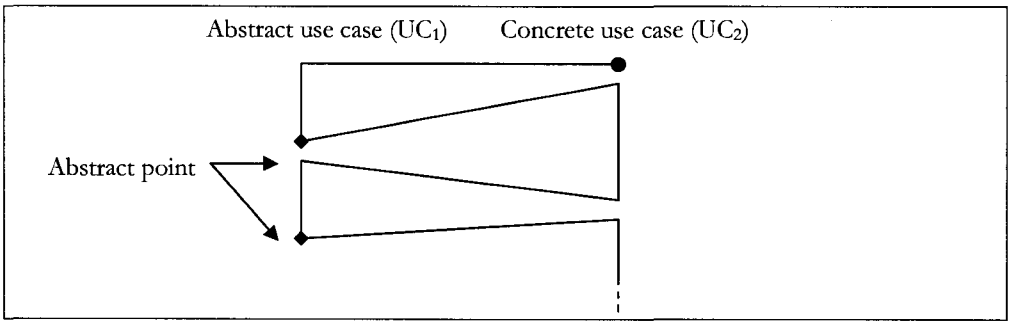


Figure 5-3: The execution path of one child use case with more than one abstract point in its parent use case

5.3.1.2 More than one child use case and only one abstract point in their parent use case

[Parent use case (abstract use case UC_1)]
Basic Course:
1. Step 1
2. *abstract point*
3. Step 3

Sub-course:
{abstract} abstract point

[Child use case (concrete use case UC_2)]
Basic course
...
Sub-course: *abstract point*
1. Step 1
2. Step 2
3. Step 3
...

[Child use case (concrete use case UC_3)]
... (*other child use cases*)

[Child use case (concrete use case UC_n)]
Basic course:
...
Sub-course: *abstract point*
1. Step 1
2. Step 2
...

Example:
[Parent use case: "Arrange Payment"]
Basic course:
1. System displays Payment method selection
2. Customer selects *Payment Method*.
3. System displays order number and record.
4. System displays Payment complete

Sub-course:
{abstract} Payment Method

[Child use case: "Credit Card Payment"]
Basic course:
...
Sub-course: *Payment Method*
1. Customer selects Credit Card Payment
2. Customer types credit card number
3. Customer types expiration date
4. System validates credit card information

[Child use case: "Cheque Payment"]
Basic course:
...
Sub-course: *Payment Method*
1. Customer selects Cheque Payment
2. System puts order on hold

Figure 5-4: An example showing more than one child use case specialized from their parent use case with only one abstract point

In the case of a parent use case with more than one specialized child use case (i.e., UC_2, UC_3,..., UC_n), if there is only one abstract point in the parent use case, a specialization condition is required to lead to one of its child use cases. In other words, the execution path must follow the flow of events in one of the child use case at a time. For instance, as we know from Order Payment knowledge, either Credit Card Payment or Cheque Payment has to be chosen when we pay the order, which means the flow of events in the use case: "Credit Card Payment" and the flow of events in the use case: "Cheque Payment" are mutually exclusive. In Figure 5-4, to the right is an example, fragments of use case descriptions, which corresponds to the general description on the left. The use case: "Arrange Payment" is specialized into two child use cases, the use case: "Credit Card Payment" and the use case: "Cheque Payment". The use-case instantiation occurs from the child use case, either one of the child use cases can be instantiated at one time. After one of the child use

cases are instantiated, the basic course of the parent use case: "Arrange Payment" will be executed. A bifurcation occurs when customers choose payment methods. This occurs when the parent use case reaches an abstract point. A specialization condition results in the execution path choosing one of the mutually exclusive flows of events. When a customer chooses the payment methods, either credit card payment method or cheque payment method can be selected. They are specialization conditions of the use case: "Credit Card Payment" and "Cheque Payment" respectively. After one of the specialization conditions is satisfied, the corresponding concrete flows of events in the child use case can be reached. Figure 5-5 is an illustration of the execution path. In brief, the control-flow construction of each pair consisting of a child use case and a parent use case can be summarized as the following steps:

- The child use case $UC_2/\ldots/UC_n$ is instantiated.
- The execution path starts from the child use case $UC_2/\ldots/UC_n$.
- The execution path executes the flow of events in the parent use case UC_1. (The child use case UC_2 does not define it but inherit it from its parent use case UC_1)
- The parent use case UC_1 reaches an abstract point.
- The execution path reaches a specialization condition to lead to the child use case $UC_2/\ldots/ UC_n$.
- As long as the specialization condition is satisfied, the execution path continues to execute the flow of events in this child use case $UC_2/\ldots/ UC_n$.
- Return to Step 1 until all child use cases corresponding to this abstract point have been reached.

In this presence of a generalization relationship, the number of the child use case is the number of the execution paths. Once a child use case UC_n is instantiated, its corresponding specialization condition will be reached at the abstract point. Therefore, in a sense, the more child use cases a parent use case have, the more execution paths there will be.

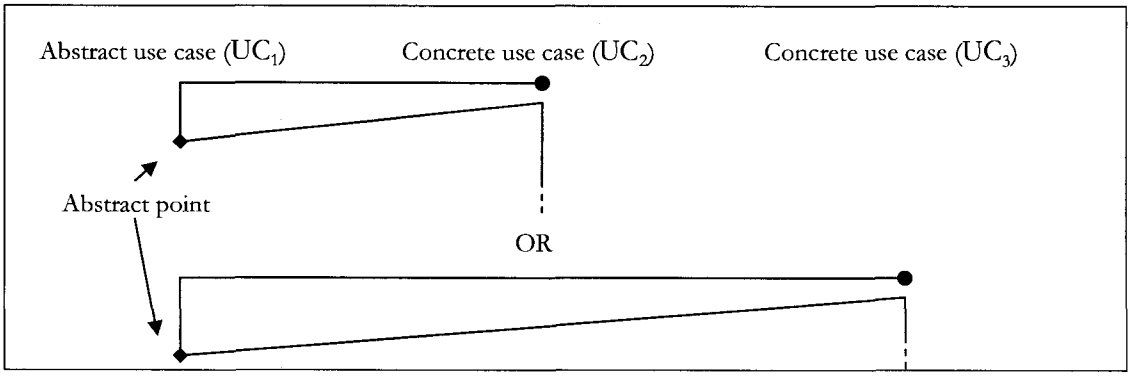


Figure 5-5: The execution path of more than one child use case with one abstract point in its parent use case

5.3.1.3 More than one child use case and more than one abstract point in the parent use case.

In the case of a parent use case with more than one specialized child use case, if there is more than one abstract point in the parent use case, a specialization condition is required to lead to one of its child use cases. However, a potential rule must be set up in advance. When an abstract point in the parent use case is reached for the second time or another same abstract point has been reached, the execution path must follow the flow of events in the child use case, which had been visited the previous time. For instance, in Figure 5-6, to the right is an example, fragments of use case descriptions, which corresponds to the general description on the left. The use case: "Arrange Payment" has an abstract point "Payment Method". Due to its alternative course, the abstract point will be reached more than once, twice in actually. If at the first attempt, the abstract point "Payment Method" is followed by executing the flow of events in the child use case: "Credit Card Payment", at the second time the flow of events in the child use case: "Credit Card Payment" should be executed and vice versa. Figure 5-7 is an illustration of the execution path.

In addition to the combination of the flow of events of a parent use case with its child use cases, the corresponding pre-conditions of the use cases are inevitably changed. As has been noted, a special condition can determine whether a child use case can be reached or not. It is done in the same way as pre-condition. A pre-condition is a prerequisite prior for starting the execution path. The bifurcation can be combined into the pre-condition of the parent use case as a new pre-condition. Thus, once the new pre-condition is satisfied, the child use case can be reached. For example, the pre-condition of the use case: "Arrange Payment" is "Order System is ON AND Customer is logged in AND Order is placed". The special condition of the use case: "Credit Card Payment" is "Credit Card Payment chosen". It is combined into the pre-condition of the use case: "Credit Card Payment" and forms a new pre-condition "Order System is ON AND Customer is logged in AND Order is placed AND Credit Card Payment chosen". The pre-condition becomes a new pre-condition of the use case: "Credit Card Payment". Similarly, the new pre-condition of the use case: "Cheque Payment" is "Order System is ON AND Customer is logged in AND Order is placed AND Cheque Payment chosen".

[Parent use case (abstract use case UC_1)]
Basic Course:
1. Step 1
2. *abstract point*
3. Step 3
4. *abstract point*

Sub-course:
{abstract} abstract point

[Child use case (concrete use case UC_2)]
Basic course
...
Sub-course: *abstract point*
1. Step 1
2. Step 2
3. Step 3
...

[Child use case (concrete use case UC_3)]
... (*other child use cases*)

[Child use case (concrete use case UC_n)]
Basic course:
...
Sub-course: *abstract point*
1. Step 1
2. Step 2
...

Example:
[Parent use case: "Arrange Payment"]
Basic course:
1. System displays Payment method selection
2. Customer selects *Payment Method*.
3. System displays order number and record.
4. System displays Payment complete

Sub-course:
{abstract} Payment Method
Alternative course:
1a. After 60 seconds
1a1. System alarm warning
1a2. GOTO 1

[Child use case: "Credit Card Payment"]
Basic course:
...
Sub-course: *Payment Method*
1. Customer selects Credit Card Payment
2. Customer types credit card number
3. Customer types expiration date
4. System validates credit card information

[Child use case: "Cheque Payment"]
Basic course:
...
Sub-course: *Payment Method*
1. Customer selects Cheque Payment
2. System puts order on hold

Figure 5-6: An example showing more than one child use case specialized from their parent use case with more than one abstract point

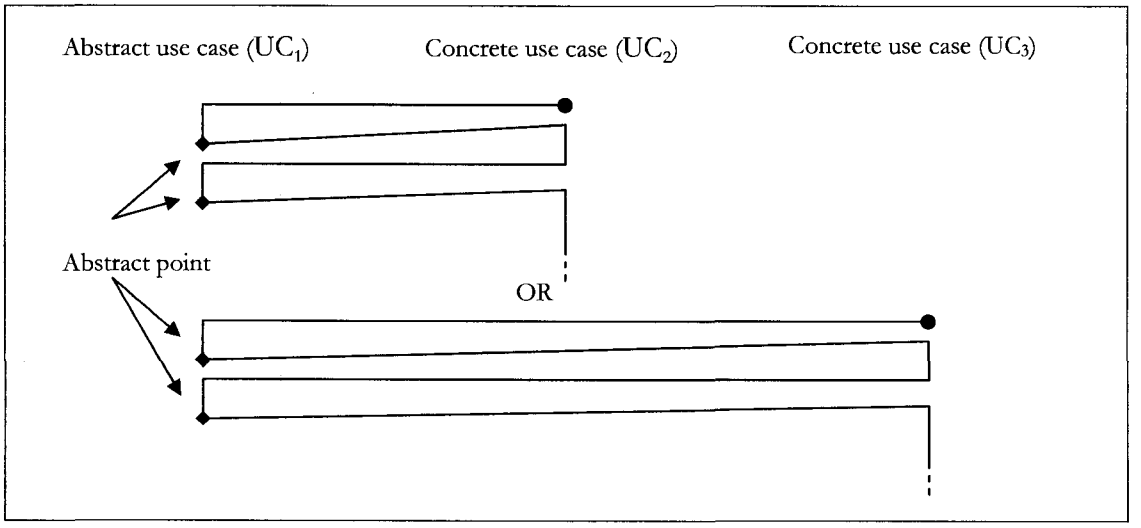


Figure 5-7: The execution path of more than one child use case with more than one abstract point in their parent use case

5.3.2 Sequential-Relation Inference from Pre-&Post-condition

In this section, we propose an approach to analyzing the sequential relations of each pair of use cases. The objective of this approach is to capture these sequential relations explicitly rather than leaving them implicitly. The explicit sequential relations will be represented as precede relations. The scope of the use cases to infer the precede relations will be introduced first. Then, the precede-relation inference based on predicates will be presented. Finally, the precede relations together with traditional UML use case relationships will be described in a global combined graph.

5.3.2.1 The Scope of Sequential-Relation Inference

As discussed previously, in order to decompose functions, traditional UML use case relationships are used in the use case model, such as include, extend and generalization. In Chapter 2, the directions of control flows among the use cases with UML use case relationships were discussed. For instance, the control flows in an included use case will be combined into its base use cases when the use-case instance reaches an inclusion directive. With regards to precede-relation inference, the sequential relations of the use cases that are connected with the UML use case relationships do not need to be considered. The use-case flow of included use cases and extending use cases will be combined into their base use cases, and use-case flow of specialized use cases will be combined into their abstract use cases (as discussed in previous sections). The use cases that we are considering to infer the precede relations are called "main use cases". In our approach, the precede-relation inference only occurs among main use cases. The definition of main use cases is proposed in Definition 5-1.

A use case is a main use case if it is not an included use case, an extending use case or a specialized use case. *UC_Set* is a set including all use cases that are represented in a use case model. For instance, in the OPSystem use case model, there are *12* use cases, thus the set *UC_Set* has *12* elements. In Figure 5-8, all main use cases are indicated by dashed arrows. There are seven elements in *M*.

Definition 5-1: The scope of sequential-relation inference

UC_n is a main use case if UC_n is not an included use case or an extending use case or a specialized use case.
More formally,
Let UC_Set = {All use cases represented in a use case model}
Let I = {All use cases that are included in other use cases}
Let E = {All use cases that are extending other use cases}
Let S = {All use cases that are specialized from other use cases}
The set of main use cases $M = (UC\_Set - I - E - S)$

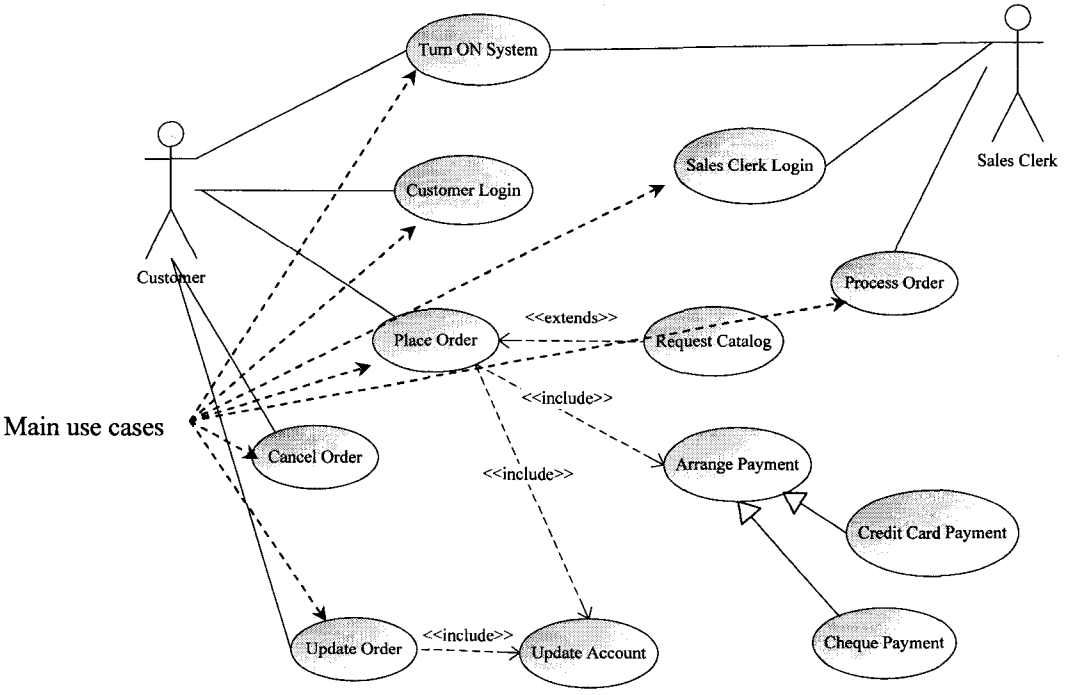


Figure 5-8: Use Case Model of Order Processing System (Main use cases are indicated)

5.3.2.2 Sequential-Relation Inference based on Predicates

As discussed in section 4.3.1, precede relations indicate that a use-case flow can be followed by a use-case flow of another use case if a precede relation exists between two use cases. Whether a use-case flow of a use case can be executed depends on whether the pre-condition of the use case holds or not. This establishes a standpoint that if pre-conditions of a use case can hold when the post-conditions of another use case hold, then a precede relation exists. Let us examine this into detail. For each use case, there is a contract, which is the pair, pre-condition and success post-condition [Sou99]. Additionally in some use cases, alternative post-conditions may appear. Due

to the existence of the two kinds of post-conditions, the precede relations can be inferred not only from pre-conditions and success post-conditions, but also from pre-conditions and alternative post-conditions. Nevertheless, not all use-case pairs have precede relations, which means not all use cases will have a succedent use case. In order to convey the inference, we propose Definition 5-2, $Post\ (UC_1) \Rightarrow\ Pre\ (UC_2)$ means when the post-conditions of UC_1 hold, the pre-conditions of UC_2 hold. The same applies on $AltPost\ (UC_1) \Rightarrow Pre\ (UC_2)$. Given two use cases, UC_1 and UC_2, there exists a sequential relation $UC_1 \times \text{precede} \times UC_2$ if $post(UC_1) \Rightarrow pre(UC_2)$ or there is an alternative condition $Altpost(UC_1)$ included in the set of alternative conditions of UC_1 such that $Altpost(UC_1) \Rightarrow pre(UC_2)$.

In section 5.2, we introduced that pre-conditions, success post-conditions and alternative post-conditions corresponding to compound-predicate can be constructed by using Boolean operators. A predicate is a pair $<E, V>$. Each non-atomic condition corresponds to a compound-predicate, which are connected with "AND" and "OR". The compound-predicate can be transformed to the form of $\bigvee_{i=1}^{m}\left(\bigwedge_{j=1}^{n} < E_j, V_j >\right)$[14], where "$\wedge$" denotes "AND" and "$\vee$" denotes "OR". The transformation conforms to logic transformation rules (distributive laws), where "$\wedge$" has a higher priority than "$\vee$". For instance, post (UC_1) = $<E_1, V_1> \wedge <E_2, V_2> \vee <E_3, V_3>$. It can be transformed to $<E_1, V_1> \wedge <E_2, V_2> \vee <E_1, V_1> \wedge <E_3, V_3>$. The transformation will separate each conjunction part $\left(\bigwedge_{j=1}^{n} < E_j, V_j >\right)$ from another conjunction part with disjunction operators. There are two threads that can lead to $post(UC_1) \Rightarrow pre(UC_2)$ or $Altpost(UC_1) \Rightarrow pre(UC_2)$. One thread contains set values within; the other thread contains no set values. Either thread can be used to infer the precede relations. In our presentation, we use P_1 to represent $post(UC_1)$ or $Altpost(UC_1)$ and use P_2 to represent $pre(UC_2)$. An example of a real use case model will be given afterwards.

[14] m is the quantifier of the conjunction parts (i.e., how many conjunction parts in a compound predicate, the number of disjunction operators plus one), n is the quantifier of the predicates in each conjunction part

Definition 5-2: The inference of sequential relations

- Given a use case $UC_1 \in M$, with the set of alternative post-conditions $Altpost(UC_1)$, and a success post-conditions $Post\ (UC_1)$
Given a use case $UC_2 \in M$, with a pre-condition $Pre\ (UC_2)$
There is a sequential relation $UC_1 \times \text{precede} \times UC_2$ between UC_1 and UC_2, IFF $post(UC_1) \Rightarrow pre(UC_2)$ OR $\exists Altpost \in Altpost(UC_1) \mid Altpost \Rightarrow pre(UC_2)$

- Given two conditions P_1 and P_2, we can show $P_1 \Rightarrow P_2$ in our context. Recall each condition is in a form like $\bigvee_{i=1}^{m}\left(\bigwedge_{j=1}^{n} < E_j, V_j >\right)$

Let $P_1 = \bigvee_{i=1}^{m}(P_{1i}), P_2 = \bigvee_{j=1}^{n}(P_{2j})$, Each $P_{1i} = \bigwedge_{k=1}^{s} < E_k, V_k >$ and each $P_{2j} = \bigwedge_{l=1}^{u} < E_l, V_l >$

- $P_1 \Rightarrow P_2$ IF
There is at least one $P_{1i} \in P_1$, and there is at least one $P_{2j} \in P_2$,
For each $< E_l, V_l > \in P_{2j}$, $\exists < E_k, V_k > \in P_{1i}$ such that $E_l = E_k$ and

- IF V_l is a set value, V_k is a set value then $V_l \subset V_k$
- IF V_l is not a set value, V_k is not a set value, $V_l = V_k$

There is a conjunction part $P_{1i} = \bigwedge_{k=1}^{s} < E_k, V_k >$ in $P_1 = \bigvee_{i=1}^{m}(P_{1i})$, and a conjunction part $P_{2j} = \bigwedge_{l=1}^{u} < E_l, V_l >$ in $P_2 = \bigvee_{j=1}^{n}(P_{2j})$. If for each predicates $< E_l, V_l > \in P_{2j}$, there exists $< E_k, V_k > \in P_{1i}$ such that $E_l = E_k$ and,

- If V_l is a set value and V_k is a set value, $P_1 \Rightarrow P_2$ is satisfied under the condition that $V_l \subset V_k$. For instance, $post(UC_1)$ = (<E_1, V_1> ∧ <E_2, V_2> ∧ <E_3, V_3>) ∨ (<E_1, V_1> ∧ <E_3, V_3> ∧ <E_4, V_4>), $pre(UC_2)$ = <E_1, V_1> ∧ <E_6, V_6> and the conjunction part in $pre(UC_2)$ is included in one of the conjunction parts <E_1, V_1> ∧ <E_2, V_2> ∧ <E_3, V_3> of $post(UC_1)$. If V_6, V_2 and V_3 are set values, then either ($V_6 \subset V_2$) and ($E_6 = E_2$), or ($V_6 \subset V_3$) and ($E_6 = E_3$) should be satisfied. Thus, a sequential relation $UC_1 \times \text{precede} \times UC_2$ is in existence. The set value is denoted by comparisons. An example of ($V_6 \subset V_2$) is that $V_6 > 2$, $V_2 > 1$ or $V_6 < 1$, $V_2 < 2$.
- If neither V_l nor V_k is a set value, $P_1 \Rightarrow P_2$ is satisfied under the condition that $V_l = V_k$. For instance, $post(UC_1)$= (<E_1, V_1> ∧ <E_2, V_2> ∧ <E_3, V_3>) ∨ (<E_1, V_1> ∧ <E_3, V_3> ∧ <E_4, V_4>), $pre(UC_2)$ = <E_1, V_1> ∧ <E_2, V_2>. If V_1, V_2 and V_3 are not set values, $pre(UC_2)$ =<E_1,

$V_1> \wedge <E_2, V_2>$ is included in one of the conjunction parts $<E_1, V_1> \wedge <E_2, V_2> \wedge <E_3, V_3>$ in $post(UC_1)$, therefore, $post(UC_1) \rightarrow pre(UC_2)$ and a sequential relation $UC_1 \times \text{precede} \times UC_2$ is in existence.

5.3.2.3 *Example of Order Process System for Sequential-Relation Inference*

In this section, we provide some examples to demonstrate Definition 5-2. Table 5-1 presents the pre-conditions and success post-conditions of the use cases in the OPSystem use case model. Table 5-2 presents the alternative post-conditions. Full use case descriptions are presented in Appendix I.

Use Case Name	Pre-condition	Post-condition
Turn ON System	OPSystem is OFF AND Customer is not logged in AND Sales Clerk is not logged in	OPSystem is ON AND Customer is not logged in AND Sales Clerk is not logged in
Customer Login	OPSystem is ON AND Customer is not logged in	OPSystem is ON AND Customer is logged in
Sales Clerk Login	OPSystem is ON AND Sales Clerk is not logged in	OPSystem is ON AND Sales Clerk is logged in
Process Order	Order is placed AND OPSystem is ON AND Sales Clerk is logged in	Order is processed AND OPSystem is ON AND Sales Clerk is logged in
Place Order	OPSystem is ON AND Customer is logged in OR Sales Clerk is logged in	Order is placed AND OPSystem is ON AND (Customer is logged in OR Sales Clerk is logged in)
Cancel Order	Customer is logged in AND OPSystem is ON AND Order is placed OR Order is on hold	Order is cancelled AND Customer is logged in AND OPSystem is ON AND
Update Order	Customer is logged in AND OPSystem is ON AND Order is placed	Customer is logged in AND OPSystem is ON AND Order is updated AND Order is placed
Credit Card Payment	OPSystem is ON AND Customer is logged in AND Order is placed	Order Payment is Complete AND OPSystem is ON AND Customer is logged in
Update Account	Customer is logged in AND OPSystem is ON	Customer account information is updated AND Customer is logged in AND OPSystem is ON
Request Catalog	None	None

Table 5-1: The pre-condition, post-condition of use cases in the use case descriptions of Order Process System

Use Case Name	Alternative condition number	Alternative Post-condition
Update Order	3a	Customer is logged in AND Order System is ON AND Order is on hold

Table 5-2: Alternative post-condition of use cases in the use case descriptions of Order Process System

- **Example 1 – Both conditions are simple conjunction parts**

The success post-condition of the use case: "Turn ON System" is

- Post (Turn ON System) = "OPSystem is ON AND Customer is not logged in AND Sales Clerk is not logged in"

The pre-condition of the use case: "Customer Login" is

- Pre (Customer Login) = "OPSystem is ON AND Customer is not logged in"

The pre-condition of the use case: "Customer Login" is included in the post-condition of the use case: "Turn ON System". Therefore, when Post (Turn ON System) is hold, the Pre (Customer Login) is hold too. The use case: "Turn ON System" has precedence in order of execution over the use case: "Customer Login". A precede relation is captured between the use case: "Turn ON System" and the use case: "Customer Login". Similarly, the use case: "Sales Clerk Login" should be executed after the use case: "Turn ON System". Another precede relation can be captured between the use case: "Turn ON System" and the use case: "Sales Clerk Login".

- **Example 2 – One condition is a simple conjunction part, the other condition is a combination of conjunction parts with disjunctions**

The success post-condition of the use case: "Customer Login" is

- Post (Customer Login) = "OPSystem is ON AND Customer is logged in"

The pre-condition of the use case: "Place Order" is

- Pre (Place Order) = "OPSystem is ON AND Customer is logged in OR Sales Clerk is logged in"

As previously discussed, Pre (Place Order) can be transformed to

- Pre' (Placer Order) = "OPSystem is ON AND Customer is logged in OR OPSystem is ON AND Sales Clerk is logged in"

One of the conjunction parts "OPSystem is ON AND Customer is logged in" in the pre-condition of the use case: "Place Order" is included in one of the conjunction parts of the success post-condition of the use case: "Customer Login". This means that when the post-condition of the use case: "Customer Login" holds, the pre-conditions of the use case: "Place Order" can hold. Therefore, a precede relation can be captured between use case: "Customer Login" and the use case: "Place Order".

♦ **Example 3 – Both conditions are a combination of conjunction parts with disjunctions**

The success post-condition of the use case: "Place Order" is

- Post (Place Order) = "Order is placed AND OPSystem is ON AND Customer is logged in OR Sales Clerk is logged in"

It can be transformed to:

- Post' (Place Order) = "Order is placed AND OPSystem is ON AND Customer is logged in OR Order is placed AND OPSystem is ON AND Sales Clerk is logged in"

The pre-condition of the use case: "Cancel Order" is

- Pre (Cancel Order) = "Customer is logged in AND OPSystem is ON AND Order is placed OR Order is on hold"

It can be transformed to:

- Pre' (Cancel Order) = "Customer is logged in AND OPSystem is ON AND Order is placed OR Customer is logged in AND OPSystem is ON AND Order is on hold"

In this case, the success post-condition of the use case: "Place Order" holds may not result in that the pre-condition of the use case: "Cancel Order" can hold. For instance, one of the conjunction parts "Order is placed AND OPSystem is ON AND Customer is logged in" in the pre-condition of the use case: "Cancel Order" is included in one of the conjunction part "Order is placed AND OPSystem is ON AND Customer is logged in" in the success post-condition of the use case: "Place Order". This means that when the post-condition of the use case: "Place Order" holds (e.g., only if the conjunction part "Order is placed AND OPSystem is ON AND Customer is logged in" holds), the pre-conditions of the use case: "Cancel Order" can hold. The success post-condition of the use case: "Place Order" can also hold if another conjunction part "Order is placed AND OPSystem is ON AND Sales Clerk is logged in" holds. In this case, it will not result in that the pre-condition of the use case: "Cancel Order" holds. However, once a possible sequential relation is detected, testing should cover this possibility. A precede relation can therefore be captured between the use case: "Place Order" and the use case: "Cancel Order".

5.3.2.4 Synchronous Sequential-Relation Inference based on Predicates

In the last section, we proposed Definition 5-2 to infer precede relations based on analyzing post-conditions and pre-conditions. With this definition in mind, given a use case UC_1, and use

case $UC_n, UC_m \in M - UC_1$, UC_n and UC_m can be executed in synchronization if $post(UC_1) \Rightarrow pre(UC_n)$ and $post(UC_1) \Rightarrow pre(UC_n)$ or there exists an alternative post-condition $Altpost(UC_1) \Rightarrow pre(UC_n)$ and $Altpost(UC_1) \Rightarrow pre(UC_m)$. For instance, Pre (UC_2) =<E_1, V_1> ∧ <E_2, V_2> ∧ <E_3, V_3>, Pre (UC_3) = <E_1, V_1> ∧ <E_2, V_2>, Post (UC_1) = <E_1, V_1> ∧ <E_2, V_2> ∧ <E_3, V_3> ∧ <E_4, V_4>. Pre (UC_2) and Pre (UC_3) are both conjunction parts and they are both included in the Post (UC_1). When the post-condition of UC_1 holds, both the pre-condition of UC_2 and UC_3 can hold. Therefore, UC_2 and UC_3 are considered to have a possibility of being executed synchronously. The synchronization of UC_2 and UC_3 is represented as a sign "||". The synchronization means that the use-case flow of two synchronization use cases can be executed in an interwoven manner. In this thesis, test scenario generation does not consider this interwoven manner.

5.3.2.5 Example of Order Process System for Synchronous Sequential-Relation Inference

The success post-condition of the use case: "Place Order" is

- Post (Place Order) = "Order is placed AND OPSystem is ON AND Customer is logged in OR Order is placed AND OPSystem is ON AND Sales Clerk is logged in"

The pre-condition of the use case: "Cancel Order" is

- Pre (Cancel Order) = "Customer is logged in AND OPSystem is ON AND Order is placed OR Customer is logged in AND OPSystem is ON AND Order is on hold"

The pre-condition of the use case: "Update Order" is

- Pre (Update Order) = "Customer is logged in AND OPSystem is ON AND Order is placed"

In this case, a conjunction part "Customer is logged in AND OPSystem is ON AND Order is placed" in the pre-condition of the use case: "Cancel Order" and the pre-condition "Customer is logged in AND OPSystem is ON AND Order is placed" of the use case: "Update Order" are both included in the conjunction part "Order is placed AND OPSystem is ON AND Customer is logged in" of the pre-condition of the use case: "Place Order". Therefore, two precede relations are captured: the use case: "Place Order" precedes the use case: "Cancel Order" and the use case: "Place Order" precedes the use case: "Update Order". In addition, the use case: "Cancel Order" and the use case: "Update Order" may be executed simutaneously.

5.3.3 Proposal of Global Combined Graph

In the previous sections, we have proposed the execution path in the presence of UML generalization relationships. Furthermore, we proposed how to infer the implicit use case sequential relations explicitly. In order to convey a clear and high-level picture of these captured use case precede relations, we will create a global combined graph based on the presentations [Arm01]. We call it a global graph because all use cases are included in this graph and all inferred sequential relations are presented as precede relations in the graph. Figure 5-9 presents a global combined graph corresponding to the OPSystem use case model. As discussed, the precede relations only appear among main use cases. The number of precede relations depends on the nature of the use case model and the content of the pre-conditions and post-conditions. Although the precede relations can be added by users manually, in some cases, it will be time-consuming and prone to flaws. An automated tool of capturing these precede relations is therefore of vital importance.

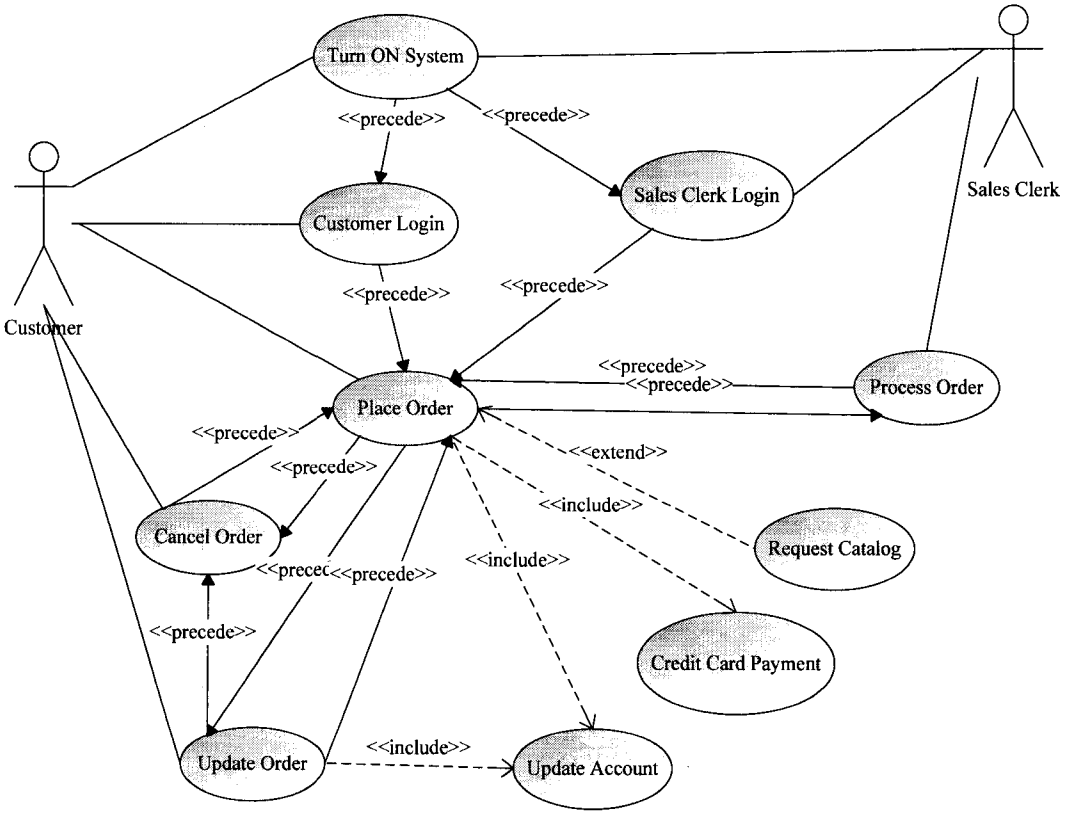


Figure 5-9: A Global Combined Graph representing OPSystem use case model

5.4 Implementation and Evaluation

In this section, we first present why a tool is required to capture sequential relations automatically. Then, we present how this approach is integrated into the UCEd framework. The overview architecture of the automated tool will be presented. Subsequently, the interface of the tool is introduced. Thereafter, we present the algorithm of designing all the functions of the tool. We also evaluate the tool to determine how its functions work with a real use case model.

5.4.1 Objective

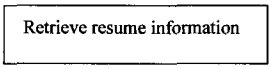
As discussed in the previous sections, precede relations are captured by analyzing pre-conditions and post-conditions based on predicates. The analysis procedure can be performed in an ad-hoc fashion. Thus, manually updating (i.e., adding and removing) precede relations are tedious and cumbersome tasks. Moreover, in some cases, the conditions with compound predicates may be composed of many conjunction parts associated with disjunctions. Comparison may involve a great deal of repetition. If the precede relations are created in a disorganized manner, the system-level test-case generation will be conducted in an ad-hoc manner, never knowing when all the precede relations are captured and tested. On the other hand, testers may test partial functions of the system under testing, which requires that specific precede relations be added. It is possible to add or remove precede relations due to different testing tasks assigned. It is rigid and with limited usage, if the automated tool does not have functions related to choosing specific precede relations. To overcome this problem, a solution of automating the precede-relation capture process is required. The solution is realized by implementing a tool, which conforms to flexible principles.

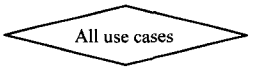
5.4.2 Architecture

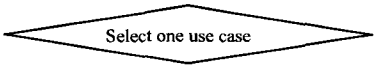
Although the automated tool for precede relations is integrated into UCEd, it is a separate component. Figure 5-10 shows the architecture of the tool. The tool is a sequential relation capture tool and we name it "Resume Operation Extractor". As discussed in Chapter 4, the "resume" operation is a directive (similar to inclusive directive) in use case descriptions meaning that a use case can precede other use cases specified in resume operations. The resume operations are regarded as separate steps in the use case descriptions. We assume that resume operation steps can only appear at the last step of a main use case description, either the last normal step or the last alternative step.

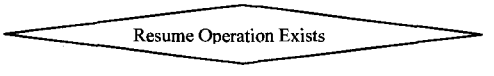
The resume operation steps cannot appear in included use cases or extending use cases. If a use case has a resume operation step with more than one resume operations, it means that it can precede more than one use cases. A user can add both precede relations in the use case model-editing panel and resume operation steps in the use case description-editing panel manually. However, both must be added. If a precede relation is added without its corresponding resume operation step added, the UCEd will report an error and vice versa. Compared with the manual updating precede relations in UCEd, this automated sequential relation capture tool is an alternative way for the users if they want to capture and update precede relations automatically.

The tool starts from loading the use case model under consideration. All main use cases are restricted for precede relation capture rather than providing all use cases. The legend description of the architecture presented in Figure 5-10 is listed below:

- The square denotes an action that users have performed or a result of system responses. For instance in the tool, 

denotes that users can perform retrieving all resume information, which is potential precede relations prepared for adding/removing from the use case model.
- The diamond denotes a branch condition that provides forks leading to different subsequent steps. The diamond can represent either the decisions that the system encounters or the actions that the user performs. For instance in the tool, 

(i.e., a user's action) denotes that user can choose either all main use cases to retrieve all resume operation information, or choose one use case to see if there are any precede relations connected to it, which is denoted by 

. The decisions that the system encounters are 

etc.
- The parallel line denotes that either of the subsequent steps can be chosen. For instance, after the user has retrieved all possible resume information, all resume and precede relations can be added or removed.

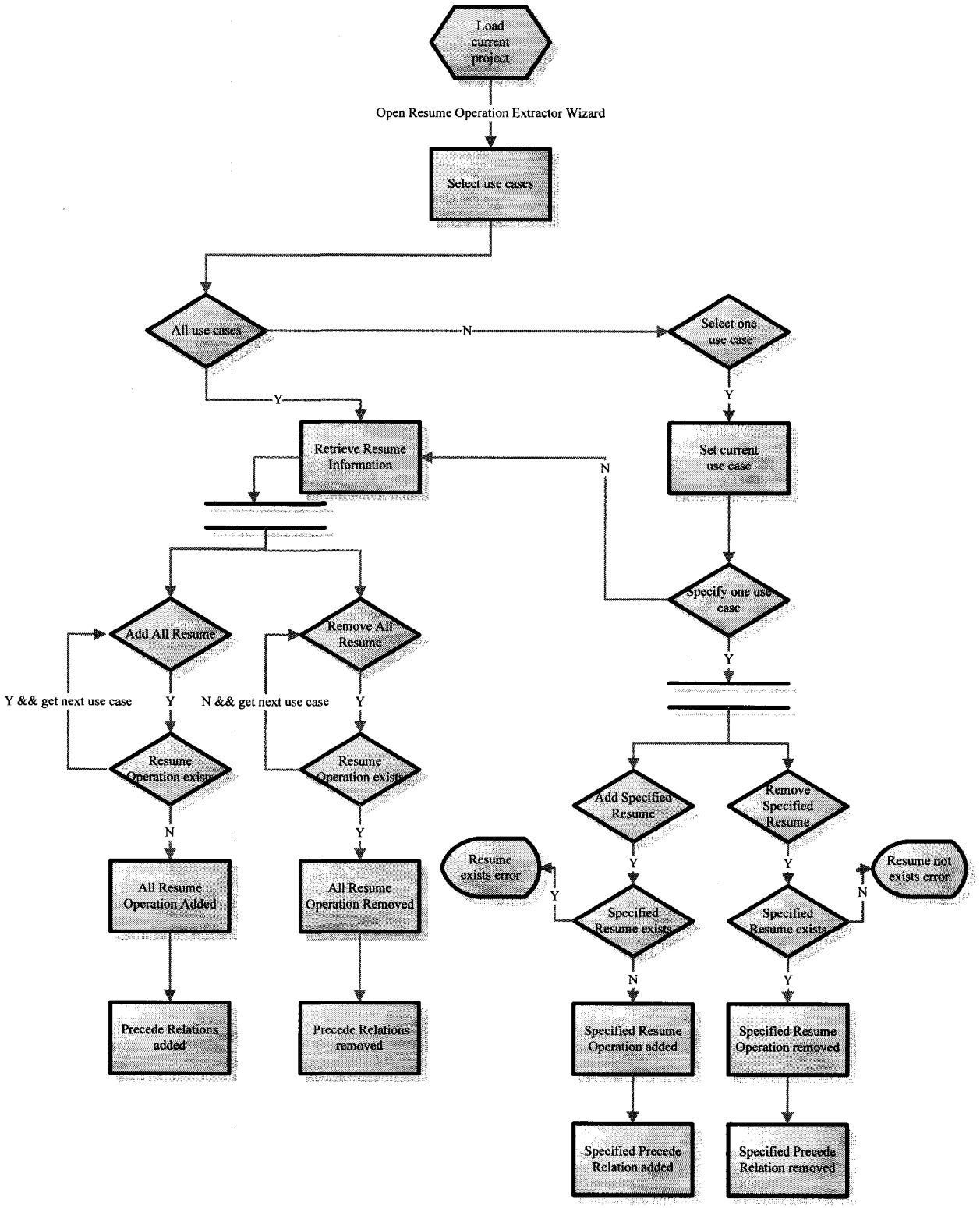


Figure 5-10: A high-level design view of the sequential relation capture tool

5.4.3 Interface

The automated sequential relation capture tool provides a simple interface. Figure 5-11 provides a view of the interface. All functions are arranged in one workbench. These functions comply with the architecture of the tool (refer to Figure 5-10). In the combo-box to the top, a user can either select all main use cases or just one of them. The main use case: "Turn ON System" is selected as the current use case under consideration. After that, the user can perform "Retrieve resume information", which provides all potential sequential relations related to the current use cases. Two potential sequential relations are listed in the list box, "Turn ON System → Customer Login" and "Turn ON System → Sales Clerk Login". If the user has previously selected all main use cases, when he/she chooses "Add All Resume", all possible resume operations as well as precede relations corresponding to all main use cases will be added to the use case model. If the user has previously selected one of the main use cases, when he/she chooses "Add All Resume", all possible resume operation as well as precede relations corresponding to the current use case will be added. In another combo-box at the bottom, the user can specify a use case in the case that he/she has already selected one of the main use cases as a current use case previously. Thus, when the user chooses "Add Specified Resume", only one resume operation, as well as a precede relation can be added to the use case model. The situation is similar when the user chooses "Remove All Resume" and "Remove Specified Resume".

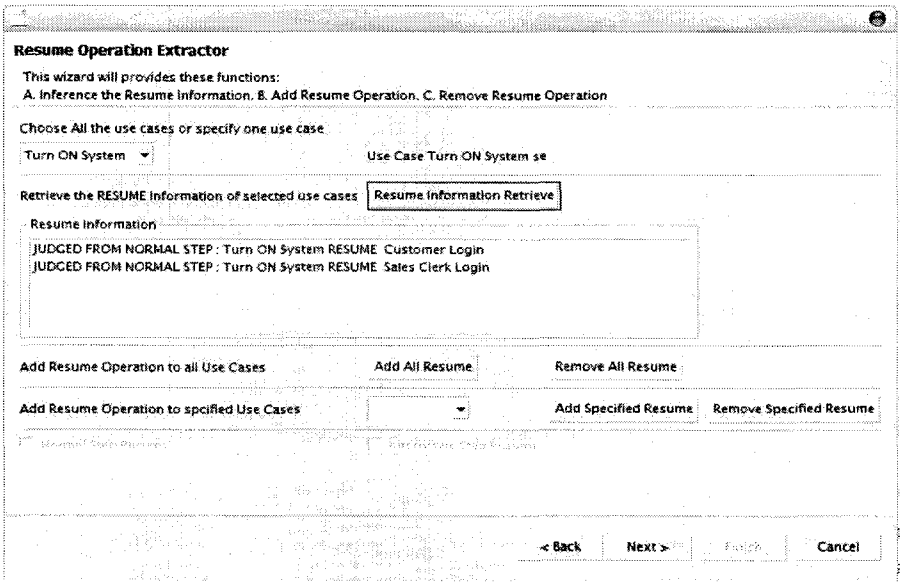


Figure 5-11: View of sequential relation capture tool, use case: "Turn ON System" is selected and two potential sequential relations are provided automatically.

5.4.4 Function and Design

In a software development life cycle (abbr., SDLC), it is not practical to complete the requirement phase before moving to the next phase as frequently use case models must be updated. That is to say, system-level test case generation will be conducted many times during the SDLC. Precede-relation capture serves system-level testing. To avoid repetitive work, it is best to adopt a flexible approach, which is to separate precede-relation extraction from precede-relation capture and separate dealing with all use cases from dealing with specific use cases. In such a flexible approach, every time the use case models are updated, new precede relations can be captured correspondingly and updated as the users desire while maintaining the generated test case compliance with system requirements.

The automated sequential relation capture tool realizes such a flexible approach by providing clear and simple click-away operations. This section will discuss the algorithms used to design these functions. For the purpose of demonstration, we have integrated them into UCEd framework. However, these algorithms, as well as the definitions proposed in previous sections, are flexible enough to be ported to other use case-based requirements with minimal changes. A performance analysis will be given in Chapter 7 by using a case study based on the tool that is built from the algorithm.

- **Functions**

The functions of the tool can be categorized as below,

A. Group precede-relation capture function
   - ♦ Capturing all potential precede relations.
   - ♦ Adding all precede relations for all main use cases.
   - ♦ Removing all precede relations from all main use cases.
   - ♦ Adding all precede relations for a current selected use case.
   - ♦ Removing all precede relations from the current selected use case.

B. Individual precede-relation capture function
   - ♦ Capturing potential precede relations for the current selected use case.
   - ♦ Adding one precede relation for the current use case to its normal scenario.
   - ♦ Adding one precede relation for the current use case to its alternative scenario.
   - ♦ Removing one precede relation from the current use case from its normal scenario.

- Removing one precede relation from the current use case from its alternative scenario.

- **Algorithm input and output**

The complete algorithms for capturing and updating resume operations as well as precede relations are represented in Figure 5-12 and Figure 5-17. The inputs and outputs of the extraction algorithm are,

- ***Input:*** A set of main uses cases from which precede relations and resume operations must be captured. An existing partial resume operations and precede relations are possible.
- ***Output:*** A required set of precede relations and corresponding resume operations of the use case model are updated for test case generation.

- **Algorithm variables**

The variables are defined as below,

- *MainUseCaseSelected*: This variable denotes that users can select all main use cases or just one of them.
- *Potential_Precede_Relations*: This variable will include all potential precede relations (implicit sequential relations), i.e., the precede relations already exist in the use case model and the ones that can be added.
- *preUC* ➔ *sucUC*: This variable denotes a potential precede relation (an implicit sequential relation).
- *preUC*×precede×*sucUC*: This variable denotes an explicit precede relation that appears on the use case model.
- *Type_of_Precede*: This variable denotes two values, one is that the precede relation is captured from success post-conditions and pre-conditions (*Type_of_Precede* = *Normal*), the other is that the precede is captured from alternative post-conditions and pre-conditions (*Type_of_Precede* = *Alternative*).
- N_Resume_{preUC}: This variable denotes a set that involves succedent use cases in the normal resume operations of a use case *preUC*.
- A_Resume_{preUC}: This variable denotes a set that involves succedent use cases in the alternative resume operations of a use case *preUC*.
- $precede_{preUC}$: This variable denotes the outgoing precede relations of the use case *preUC*.

- ✧ *UserAction*: This variable denotes the actions that users make on the tool. The value can be "Retrieve_Potential_Precede_Relations", "Add_All_Precede", "Remove_All_Precede", "Add_Specified_Precede", "Remove_Specified_Precede", "Alternative_Option_Selected", or "Normal_Option_Selected".

5.4.4.1 Design of Group Precede-Relation Capture Function

As presented in the previous section, the group precede-relation capture function has two types: "All main use case" selected and "One main use case" selected.

A use case consists of use case descriptions. Use case descriptions processed by the algorithm are the use case pre-conditions(s), use case post-condition(s), use case steps and use case alternative steps. In the algorithm presented in Figure 5-12, the function **List_Potential_Precede_Relations** (*MainUseCaseSelected*) is used to retrieve all potential precede relations. This function would be useful for the user to consider which precede relations are necessary to generate test cases. Step A1 relates to the processing of all main use cases selected (*MainUseCaseSelected* = *ALL*). Step A1.1 and A1.2 are two FOR loops to compare each use-case pair. In Step A1.1, we get the success post-condition of *preUC*, and we get the alternative post-condition (A1.1.1) for each alternative in the use case description of *preUC* so that each alternative can be reached to achieve the applicable alternative post-conditions. In Step A1.2, we get the pre-condition of *sucUC*. The next step is to analyze each pair of success post-conditions and pre-conditions. Step A1.2.1 relates to ensure that neither of the success post-condition and pre-condition is empty. According to Definition 5-1, for each use-case pair *preUC* and *sucUC*, the existence of sequential relations is based on the theory that success post-conditions of *preUC* hold, the pre-conditions of *sucUC* can hold as well (Post (*preUC*) $\Rightarrow$ Pre (*sucUC*)). Furthermore, according to Definition 5-2, each conjunction part of the pre-conditions of *sucUC* will be compared to each conjunction part of the success post-conditions of *preUC*. Then, the potential precede relations will be added to the list *Potential_Precede_Relations*. Step A1.2.2 relates to infer the sequential relations from alternative post-conditions and pre-conditions. Step A2 relates to processing of only one main use case selected (*MainUseCaseSelected* $\neq$ *ALL*). The current selected use case *curUC* will be compared with other use cases. The sequential relations will be inferred between the success post-conditions, every alternative post-conditions of the *curUC* and the pre-conditions of other use cases.

List_Potential_Precede_Relations (*MainUseCaseSelected*):*Potential_Precede_Relations*
//init: *Potential_Precede_Relations* = Ø
A1 IF (*MainUseCaseSelected* = ALL), THEN
 A1.1 FOR each use case ∈ *M*, name it *preUC*, Post (*preUC*) ← success post-condition of *preUC*
  A1.1.1 FOR each alternative of *preUC*, AltPost (*preUC*) ← alternative post-condition of *preUC*
 A1.2 FOR each use case ∈ *M*, name it *sucUC*, Pre (*sucUC*) ← pre-condition of *sucUC*
  A1.2.1 IF (Pre (*sucUC*) ≠ Ø && Post (*preUC*) ≠ Ø) && (Post (*preUC*) ⇒ Pre (*sucUC*)), THEN
   A1.2.2.1 *Potential_Precede_Relations* = *Potential_Precede_Relations* ∪ {*preUC* ➔ *sucUC*};
  A1.2.2 IF (Pre (*sucUC*) ≠ Ø && AltPost (*preUC*) ≠ Ø) && (AltPost (*preUC*) ⇒ Pre (*sucUC*)), THEN
   A1.2.3.1 *Potential_Precede_Relations* = *Potential_Precede_Relations* ∪ {*preUC* ➔ *sucUC*};
A2 ELSE IF (*MainUseCaseSelected* ≠ ALL), THEN
 A2.1 *curUC* = *UseCaseSelected*;
 A2.2 FOR each alternative of *curUC*, AltPost (*curUC*) ← alternative post-condition of *curUC*
 A2.3 FOR each use case ∈ *M*, name it *sucUC*, Pre (*sucUC*) ← pre-condition of *sucUC*
  A2.3.1IF (Pre (*sucUC*) ≠ Ø && Post (*curUC*) ≠ Ø) && (Post (*preUC*) ⇒ Pre (*curUC*)), THEN
   Potential_Precede_Relations = *Potential_Precede_Relations* ∪ {*curUC* ➔ *sucUC*};
  A2.3.2 IF (Pre (*sucUC*) ≠ Ø && AltPost (*preUC*) ≠ Ø) && (AltPost (*preUC*) ⇒ Pre (*sucUC*)), THEN
   Potential_Precede_Relations = *Potential_Precede_Relations* ∪ {*curUC* ➔ *sucUC*};
A3 return *Potential_Precede_Relations*;

Add_A_Precede_Relation (*Type_of_Precede*, *preUC*, *sucUC*)
B1. IF *Type_of_Precede* is *Normal*, THEN
 B1.1. IF (*sucUC* ∉ N_Resume_{preUC})
  B1.1.1 N_Resume_{preUC} = N_Resume_{preUC} ∪ {*sucUC*};
  B1.1.2 $Precede_{preUC}$ = $Precede_{preUC}$ ∪ { *preUC*×precede×*sucUC* };
B2. ELSE IF *Type_of_Precede* is *Alternative*, THEN
 B2.1 IF (*sucUC* ∉ A_Resume_{preUC})
  B2.1.1 A_Resume_{preUC} = A_Resume_{preUC} ∪ {*sucUC*};
  B2.1.2 $Precede_{preUC}$ = $Precede_{preUC}$ ∪ { *preUC*×precede×*sucUC*}

Remove_A_Precede_Relation (*Type_of_Precede*, *preUC*, *sucUC*)
C1. IF *Type_of_Precede* is *Normal*, THEN
 C1.1. IF (*sucUC* ∈ N_Resume_{preUC})
  C1.1.1 N_Resume_{preUC} = N_Resume_{preUC} - {*sucUC*};
  C1.1.2 $Precede_{preUC}$ = $Precede_{preUC}$ - { *preUC*×precede×*sucUC* }
C2. ELSE IF *Type_of_Precede* is *Alternative*, THEN
 C2.1 IF (*sucUC* ∈ A_Resume_{preUC})
  C2.1.1 A_Resume_{preUC} = A_Resume_{preUC} -{*sucUC*};
  C2.1.2 $Precede_{preUC}$ = $Precede_{preUC}$ -{ *preUC*×precede×*sucUC* }

Updating_Group_Precede_Relations (*MainUseCaseSelected*, *UserAction*)
D1 IF (*MainUseCaseSelected* = ALL) && (*UserAction* = Retrieve_Potential_Precede_Relations), THEN
 D1.1 *Potential_Precede_Relations* = **List_Potential_Precede_Relations** (*MainUseCaseSelected*);
 D1.2 FOR each use case UC_1 in UC_1 ➔ UC_2 ∈ *Potential_Precede_Relations*,
  D1.2.1 For each use case UC_2 in UC_1 ➔ UC_2 ∈ *Potential_Precede_Relations*,
  D1.2.2 IF (*UserAction* = Add_All_Precede), THEN
   D1.2.2.1 **Add_A_Precede_Relation** (*Type_of_Precede*, UC_1, UC_2);
  D1.2.3 ELSE IF (*UserAction* = Remove_All_Precede), THEN
   D1.2.3.1 **Remove_A_Precede_Relation** (*Type_of_Precede*, UC_1, UC_2);
D2 ELSE IF (*MainUseCaseSelected* ≠ ALL) && (*UserAction* = Retrieve_Potential_Precede_Relations), THEN
 D2.1 *Potential_Precede_Relations* = **List_Potential_Precede_Relations** (*MainUseCaseSelected*);
 D2.2 *curUC* = *MainUseCaseSelected*;
 D2.3 For each use case UC_2 in *curUC* ➔ UC_2 ∈ *Potential_Precede_Relations*

D2.3.1 IF (*UserAction* = Add_All_Precede), THEN
D2.3.1.1 **Add_A_Precede_Relation** (*Type_of_Precede, curUC, UC*$_2$);
D2.3.2 ELSE IF (*UserAction* = Remove_All_Precede), THEN
D2.3.2.1 **Remove_A_Precede_Relation** (*Type_of_Precede, curUC, UC*$_2$);

Figure 5-12: Algorithm of automatically capturing and updating (adding/removing) group precede relations

The function **Updating_Group_Precede_Relations** (*MainUseCaseSelected, UserAction*) is used to update the precede relations. The users can refer to the potential precede-relation list to update a group of precede relations (Step D1.1). If all main use cases are selected (*MainUseCaseSelected* = *ALL*) and when the user has performed Add_All_Precede (D1.2.2) or Remove_All_Precede (D1.2.3), all potential precede relations will be updated in the use case model (D1.2.2.1, D1.2.3.1). If only one main use case is selected (*MainUseCaseSelected* ≠ *ALL*), only the all potential precede relations that fire from this main use case will be updated (D2.3.1.1, D2.3.2.1). The function **Add_A_Precede_Relation** (*Type_of_Precede, preUC, sucUC*) is used to add a precede relation. Steps B1 and B2 relate to adding the precede relations and resume operations under the condition that they are not already in existence. The resume operations will be added to the use case descriptions and the precede relation will be added to the use case model. If the *Type_of_Precede* is *normal* (B1) and N_Resume$_{preUC}$ does not have the *sucUC* (B1.1), the use case *sucUC* will be added to the normal resume operation of the use case *preUC* (B1.1.1). A new precede relation *preUC*×precede×*sucUC* will be added to the use case model (B1.1.2). If the *Type_of_Precede* is *Alternative* (B2) and A_Resume$_{preUC}$ does not have the *sucUC* (B2.1), the use case *sucUC* will be added to the corresponding alternative resume operation of the use case *preUC* (B2.1.1). A new precede relation *preUC*×precede×*sucUC* will be added to the use case model (B2.1.2). The function **Remove_A_Precede_Relation** (*Type_of_Precede, preUC, sucUC*) is used to remove a precede relation. Step C1 and C2 relate to removing the precede relations and resume operations under the condition that they are already in existence. The resume operations will be removed from the use case description and the precede relation will be removed from the use case model. If the *Type_of_Precede* is *normal* (C1) and N_Resume$_{preUC}$ has the *sucUC* (C1.1), the use case *sucUC* will be removed from the normal resume operation of the use case *preUC* (C1.1.1). The corresponding precede relation *preUC*×precede×*sucUC* will be removed from the use case model (C1.1.2). If the *Type_of_Precede* is *Alternative* (C2) and A_Resume$_{preUC}$ has the *sucUC* (C2.1), the use case *sucUC* will be removed from the alternative resume operation of the use case *preUC* (C2.1.1). The corresponding precede relation *preUC*×precede×*sucUC* will be removed from the use case model (C2.1.2). Some screenshots are provided to illustrate these functions afterwards.

- **User has selected "ALL main use cases"**

Figure 5-13 shows a screenshot of the tool. All main use cases are selected and all potential precede relations of the OPSystem use case model are listed. Figure 5-14 shows a screenshot of the result after adding all precede relations. The precede relations are shown in the use case model-editing panel. The resume operations are shown in the corresponding use case description-editing panel.

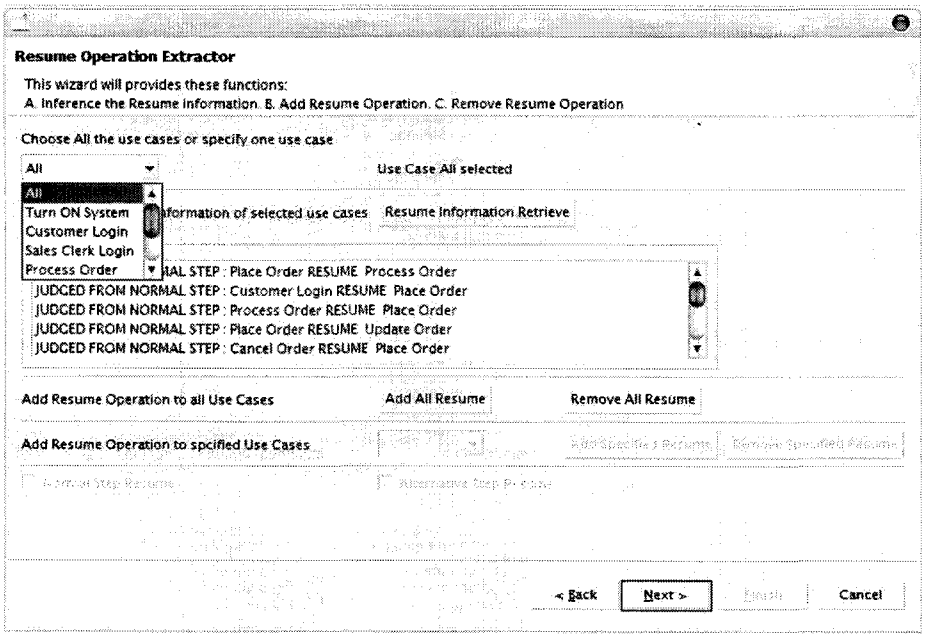


Figure 5-13: A view of the tool, ALL main use cases are selected and potential precede relations are listed

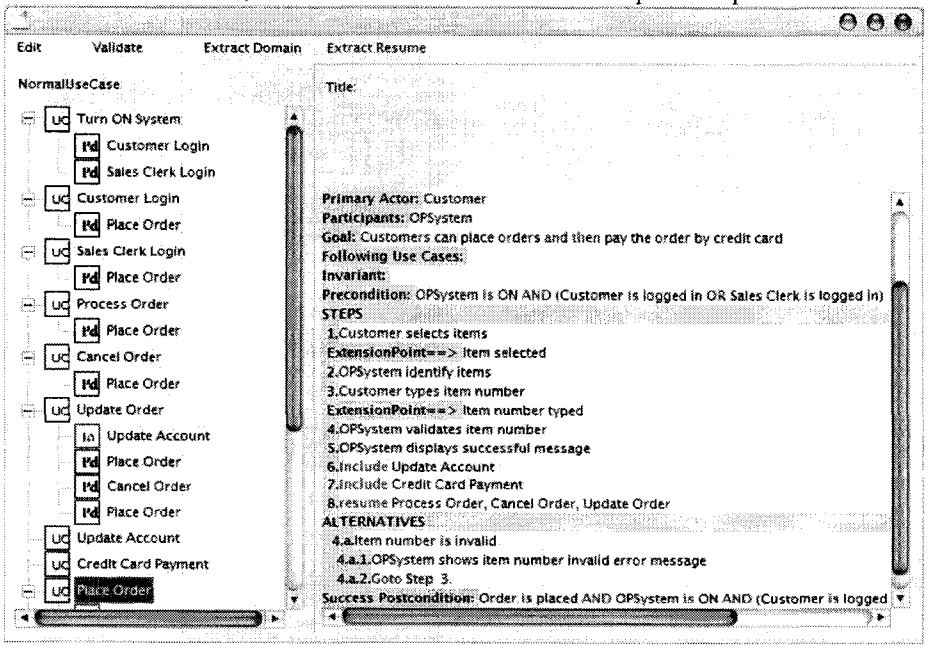


Figure 5-14: A view of use case model-editing panel. The results after executing Add All Resume in the tool

- **User has selected "One main use case"**

Figure 5-15 shows a screenshot of the tool. The main use case: "Place Order" is selected and all potential precede relations of the use case: "Place Order" are listed. Figure 5-16 shows a screenshot of the result after adding all precede relations. The precede relations are shown in the use case model-editing panel. The resume operations are shown in the corresponding use case description-editing panel.

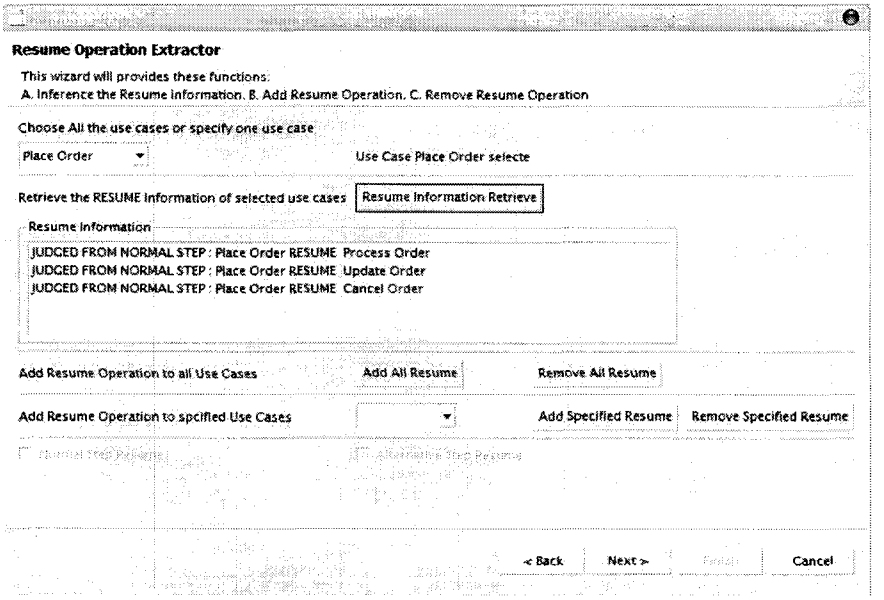


Figure 5-15: A view of the tool, one main use case: "Place Order" is selected and three potential precede relations are provided automatically.

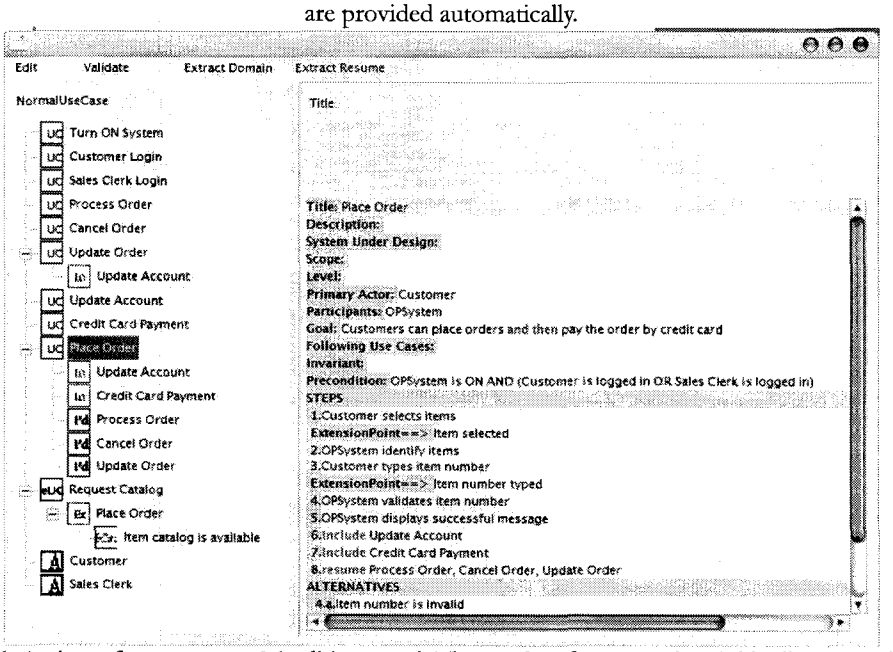


Figure 5-16: A view of use case model-editing panel. The results after executing Add All Resume in the tool

5.4.4.2 Design of Individual Precede-Relation Capture Function

As mentioned, users may want to add an individual precede relation at one time. Each precede relation connects two use cases. We presented that the user can select one main use case in the previous section. In order to provide an individual precede relation capture, the user should confirm a specified use case as the succedent use case. Figure 5-17 presents the algorithm for updating an individual precede relation.

Updating_Individual_Precede_Relation (*MainUseCaseSelected*≠ALL, *UserAction*, *SpeUCSelected*)
E1. *curUC* = *MainUseCaseSelected*;
 E1.1 IF (*UserAction* = Retrieve_Potential_Precede_Relations)
  E1.1.1 *Potential_Precede_Relations* = **List_Potential_Precede_Relations** (*curUC*);
E2. *speUC* = *SpeUCSelected*; //*curUC*➔*speUC* ∈ *Potential_Precede_Relations*
 E2.1 IF (Post (*curUC*) ⇒ Pre (*speUC*)) && (AltPost(*curUC*) ⇒ Pre(*speUC*))
  E2.1.1 Alternative Option is enabled && Normal Option is enabled
  E2.1.2 IF (*UserAction* = Add_Specified_Precede), THEN
   E2.1.2.1 IF (*UserAction* = Alternative_Option_Selected), THEN
    - **Add_A_Precede_Relation** (*Alternative*, *curUC*, *speUC*);
   E2.1.2.2 IF (*UserAction* = Normal_Option_Selected), THEN
    - **Add_A_Precede_Relation** (*Normal*, *curUC*, *speUC*);
   E2.1.2.3 ELSE ERROR
  E2.1.3 IF (*UserAction* = Remove_Specified_Precede), THEN
   E2.1.3.1 IF (*UserAction* = Alternative_Option_Selected), THEN
    - **Remove_A_Precede_Relation**(*Alternative*, *curUC*, *speUC*);
   E2.1.3.2 ELSE IF (*UserAction* = Normal_Option_Selected), THEN
    - **Remove_A_Precede_Relation**(*Normal*, *curUC*, *speUC*);
 E2.2 ELSE IF Post (curUC) ⇒ Pre(sucUC), THEN
  E2.2.1 Normal Option is enabled
  E2.2.2 IF (*UserAction* = Add_Specified_Precede), THEN
   E2.2.2.1 IF (*UserAction* = Normal_Option_Selected), THEN
    - **Add_A_Precede_Relation** (*Normal*, *curUC*, *speUC*);
   E.2.2.2.2 ELSE ERROR
  E2.2.3 ELSE IF (*UserAction* = Remove_Specified_Precede), THEN
   E2.2.3.1 IF (*UserAction* = Normal_Option_Selected), THEN
    - **Remove_A_Precede_Relation**(*Normal*, *curUC*, *speUC*);
   E2.2.3.2 ELSE ERROR
 E2.3 ELSE IF AltPost(*curUC*) ⇒ Pre(*sucUC*), THEN
  E2.3.1 Alternative Option is enabled
  E2.3.2 IF (*UserAction* = Add_Specified_Precede), THEN
   E2.3.2.1 IF (*UserAction* = Alternative_Option_Selected), THEN
    - **Add_A_Precede_Relation** (*Alternative*, *curUC*, *speUC*);
   E2.3.2.2 ELSE ERROR
  E2.3.3 ELSE IF (*UserAction* = Remove_Specified_Precede), THEN
   E2.3.3.1 IF (*UserAction* = Alternative_Option_Selected), THEN
    - **Remove_A_Precede_Relation**(*Alternative*, *curUC*, *speUC*);
   E2.3.3.2 ELSE ERROR

Figure 5-17: Algorithm of automatically updating (adding/removing) an individual precede relation

This algorithm begins with providing all potential precede relations fired from the selected main use case. The user can then specify a use case that he/she wants to constitute to a precede relation with the main use case (E2). The specified use case must refer to the listed potential precede relations. Step E2.1.1 relates to notifying the user that both an alternative option and a normal option can be selected (i.e., they are highlighted). This situation is under the condition that the specified use case can follow the main use case from both normal steps (i.e., Post (*curUC*) $\Rightarrow$ Pre (*speUC*)) and alternative steps (i.e., AltPost (*curUC*) $\Rightarrow$ Pre (*speUC*)). Either the normal option or the alternative option must be selected before the precede relation and resume operation can be added to the use case model and use case description (Step E2.1.2.1, Step E2.1.2.2). The existent precede relation and resume operation can be removed and either of the two options should be selected in advance (E2.1.3.1, E2.1.3.2). Step E2.2.1 relates to notifying the user only the normal option and Step E2.3.1 relates to notifying the user only the alternative option. An example will be used to evaluate this function in the next section.

5.4.4.3 Evaluation of Individual Precede-Relation Capture Function

As noticed, if a user selects "ALL" main use cases, the button "Add Specified Resume" and "Remove Specified Resume" will keep being disabled to avoid wrong operations (Figure 5-18).

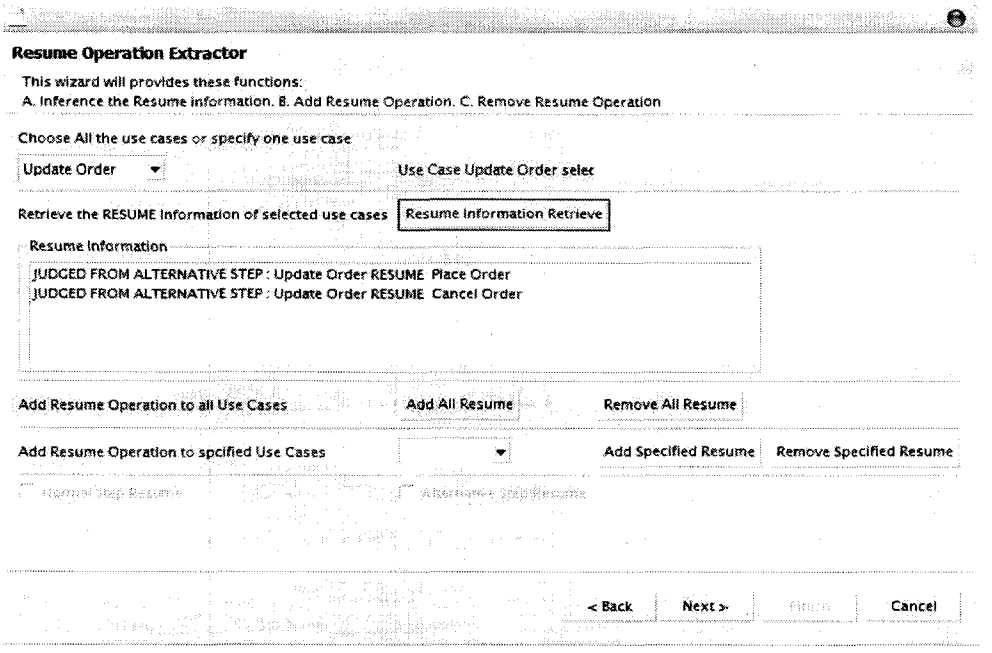


Figure 5-18: A view of sequential relation capture tool, the main use case Update Order is selected and two potential precede relations prompt

When the user has specified a use case to follow the selected main use case, the two buttons will be enabled so that user can make an individual precede-relation update. In addition, a main use case may precede a specified use case either from main success scenarios or from alternative scenarios. To deal with this case, a restrict mechanism is provided to avoid wrong operations. For instance, Figure 5-19 presents a view of the tool. The user has chosen the main use case: "Place Order" and three potential precede relations are shown. All of the three potential succedent use cases: "Process Order", "Update Order", "Cancel Order" are resumed from the normal step of the main use case: "Place Order" by comparing their success post-conditions and pre-conditions. Thus, no matter which succedent use case the user specifies, only the check box of "Normal Step Resume" is enabled. Figure 5-20 presents another view of the tool, the user has selected the main use case: "Update Order" and two potential precede relations are shown. Both of them are resumed from the alternative steps of the main use case: "Place Order" by comparing alternative post-conditions and pre-conditions. Thus, no matter which use case the user chooses, only the check box of "Alternative Step Resume" is enabled. Thanks to this restricted mechanisms, the sequential relation capture tool provides an easy-to-use function for the user to update the precede relations.

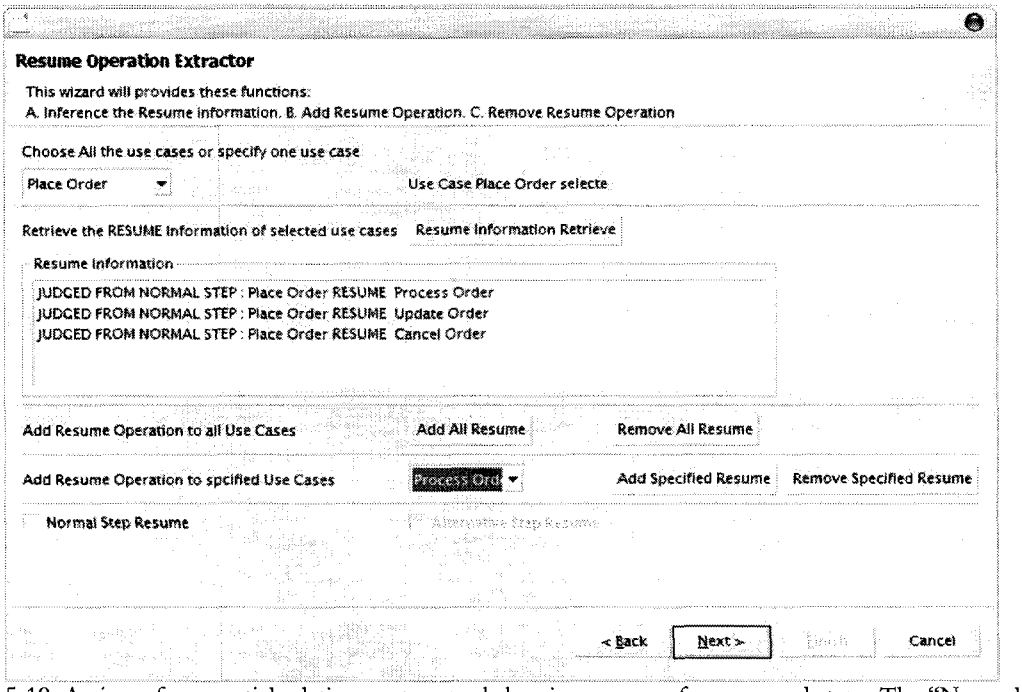


Figure 5-19: A view of sequential relation capture tool showing resumes from normal steps. The "Normal Step Resume" option is enabled while the "Alternative Step Resume" option is disabled

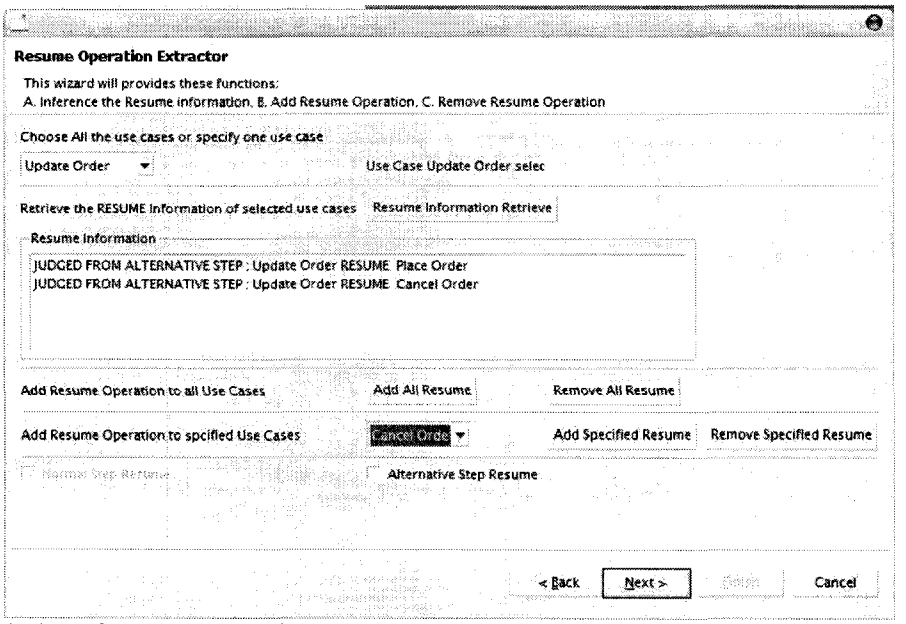


Figure 5-20: A view of the sequential relation capture tool showing resumes from alternative steps. The "Normal Step Resume" option is disabled while the "Alternative Step Resume" option is enabled

In addition to the automated updating precede relations, the tool provides the user real-time interaction dialog boxes for an easy and reliable usage. If the selected main use case does not have succedent use cases added, when the user attempts to add a succedent use case to constitute a precede relation by pressing the "Add Specified Resume" button, a successful message will be shown (Figure 5-21). However, the user cannot add the same precede relation twice, otherwise an error message will be shown (Figure 5-22).

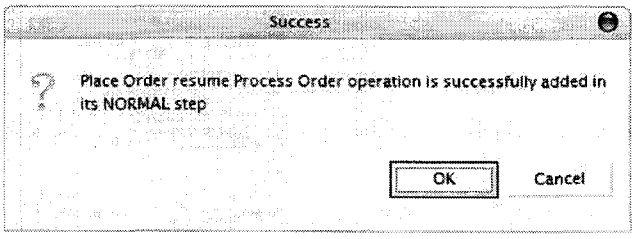


Figure 5-21: Success precede relation adding information

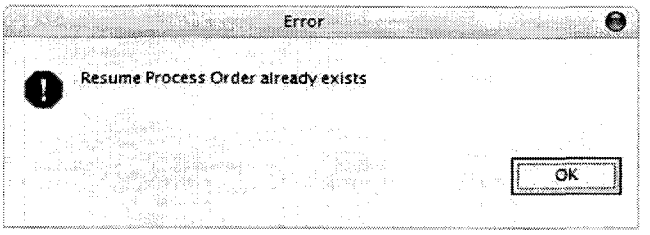


Figure 5-22: Error precede relation adding information

If a precede relation already exists in the selected main use case, an inquiry dialog box will be shown to the user with the existent use cases on the dialog box. For instance, if the selected main use case: "Place Order" already has precede relations with the use case: "Process Order" and "Cancel Order", when the user wants to add a precede relation with the use case: "Update Order", an inquiry dialog box will be shown (Figure 5-23, left). In the same way, when the specified use case resume operation will be added to the alternative resume operation step, if the selected main use case already has it, an inquiry box will be shown as well (Figure 5-23, right).

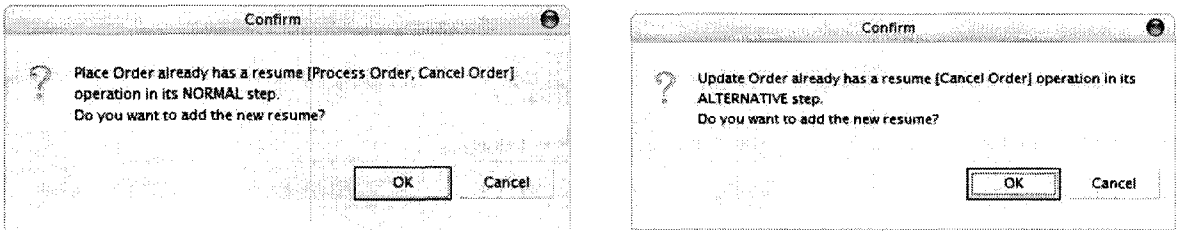


Figure 5-23: If the selected main use case has resume operations and precede relations in existence, an inquiry box will be shown prior to adding new resume operations and precede relations

5.5 Chapter Summary and Highlights

In this chapter, we first proposed a solution to solve the problem that results from UML use-case generalization relationships. The direction of execution paths from parent use cases to child use cases is pre-defined; hence, to some extent, the parent use cases and child use cases can be grouped together from the control-flow viewpoint. The execution path will automatically traverse among abstract flows and concrete flows according to the rules that we defined. An approach to generating CFSMs obtained from a use case model involving generation relationships will be proposed in the next chapter.

After that, the approach to solving the limitation of UML use case relationships was presented. Sequential relations are considered among the use cases whose use-case flows are not included in other use cases, extending other use cases or specializing other use cases. We call them main use cases. The success post-conditions and alternative post-conditions of one main use case are compared with the pre-conditions of the other main use case. These conditions are all represented as predicates. The comparison is conducted through whether one of the conjunction parts is included in the other one, which will eventually infer that when the success post-condition (or an alternative post-condition) of one main use case holds, the pre-condition of another main use case

will hold as well. Thus, a precede relation yields.

As noted, the process of inferring precede relations are repetitive, which means an automated way of conducting the inference will be timesaving. In order to implement the automated approach, we integrate it into UCEd. The precede relations among use cases have a corresponding representation "resume" operation in use case descriptions. The tool that has been implemented can retrieve all resume operations which are actually potential precede relations. The precede-relations and resume operations are updated simultaneously with just one click. Compared with the arbitrary updating before the process is automated, it provides flexibility and reliability.

Chapter 6 –TEST SCENARIOS GENERATION FROM USE CASES

6.1 Introduction

Thanks to Jacobson [Jac92] and his adoption of use cases in UML, use cases have proliferated in the software engineering industry as a means of capturing user requirements and they indicate to stakeholders what a system is expected to accomplish [OMG03b]. These user requirements convey the interactions between users (i.e., information supplied by the users) and a system (i.e., expected response from the system). However, if a use case is considered as a specification of interactions to generate test cases, questions may occur, such as,

1. *What interactions have common attributes and how do they achieve a same goal?*
2. *What interactions should be captured and used to generate test cases?*

Interactions are composed of flows of events, which are considered as a source of generating test cases. The flows of events in each use-case instance realize different functionalities of a system. The functionalities can also contain sub-functionalities. This would create secondary-level use cases, such as extending use cases, included use cases and specialized use cases. The secondary-level use cases and their associated high-level use cases are subject to the same goal. The approach presented in the previous chapter sets the stage to generate test cases on a system level. Users can generate test cases aimed at different functionalities of a system by arranging different use cases together. For instance, sequential relations are inferred among different main use cases. Their secondary-level use cases will be considered as auxiliary use cases, which means the interactions of these auxiliary use cases will be captured automatically when the interactions of their main use cases are captured. Hence, the first question is answered. A main use case and its auxiliary use cases have the same goal and main use cases that have sequential relations can also be considered capable of achieving the same goal.

In this chapter, we propose an approach to generating test cases from use cases. We employ control flow-based state machines (abbr., CFSM) as intermediate models. Path sequences are then captured from traversing the control flow-based state machines by applying DFS (depth-first search) algorithms. The captured path sequences are based on certain test coverage. It would be more practical to apply test coverage in real testing as the test cases that satisfy certain coverage have the same goal or same functionality. Therefore, the test coverage answers the second question because

the interactions that are involved in these path sequences have already been adequate to generate effective test cases. Subsequently, the path sequences will evolve to test scenarios. Test scenarios are used as a basis to generate test cases. However, the inference from test scenarios to concrete test cases is not automated. Finally, we implement the approach as another component in UCEd besides the sequential relation capture tool.

6.2 Description of Path Sequence Generation Algorithm

CFSMs are obtained from parsing use cases. In previous research [Som03] [Som04b], a CFSM-generation algorithm was proposed. However, the algorithm does not take into account generalization relationships. In this section, we will not introduce how to generate the CFSMs. Instead, we will introduce the algorithm to augment the CFSMs by considering the generalization relationships.

6.2.1 Consideration of Generalization Relationship

In section 5.3.1, we presented three different situations of generalization relationships concerning the direction of execution paths. In order to obtain path sequences from use cases in the presence of generalization relationships, the original algorithm of creating CFSMs requires modification. A new algorithm considering generalization relationships is presented in Figure 6-1. The inputs and outputs of this algorithm are:

- ***Input:*** A CFSM generated from a main use case considering its extending use cases, included use cases, all general use cases attached to the main use case or included use case and all specialization conditions leading to different special use cases
- ***Output:*** A CFSM that is augmented from the original one. Abstract steps in general use cases are replaced by concrete steps in their specialized use cases.

The algorithm describes that, in a control flow-based state machine CFSM= [*E, N*] where *E* represents edges and *N* represents nodes, there is a general use case *GUC*. The original abstract step (i.e., abstract point) in the *GUC*, which is represented as an edge $n_i \times ev_i \times n_j$ in the CFSM, will be removed. If there are several special use cases specialized from the *GUC* and they correspond to this abstract step, each specialized use case will have a specialization condition. Therefore, the edge

representing the abstract step will be replaced by a new edge starting from n_i. If the special use case under consideration is SUC_n, the new edge will be $(n_i \times SC_n \bigwedge_{m=1}^{n-1} NOT(SC_m) \bigwedge_{m=n+1}^{m=k} NOT(SC_m) \times n_c)$. It indicates that only the specialization condition for the SUC_n is satisfied. The rest of the nodes and edges according to the steps in the SUC_n will be added to the n_c. After the execution path reaches the end of the steps in the SUC_n, which are the end of main success scenario or the failure scenario, it will jump back to its general use case *GUC*. To sum up, the execution path will follow the concrete flow of events in a special use case under its specialization condition when it reaches an abstract point in its general use case. After the concrete flow of events are executed, the rest of the flow of events in the general use case will be executed. Recall in section 5.3.1, we introduced the possibility of having two same abstract steps (or, an abstract step being visited twice) in a general use case. In that case, the second time that the abstract step is reached; it should lead to the same special use case that has previously been visited.

Consider_Generalization (CFSM = [*E, N*], *GUC* = [*Title, Pre, Post, Steps*], relation = [SC_n])
//general use case can appear as main use cases, included use case, but not extending use case
//CFSM: Control flow-based state machine, *GUC*: General use case, *SC*: Special condition,
//*SUC*: Specialized use case
For each general use case *GUC*
F1 Let $n_i \times ev_i \times n_j$ be the edge of abstract step in *GUC*, ev_i is an abstract event.
F2 remove all edges $n_i \times ev_i \times n_j$ starting from n_i
F3 For all specialize use cases $SUC_n = [Title_n, Pre_n, Post_n, Steps_n] \in SUC$ that corresponds to ev_i
$\{\forall SUC_n \in SUC, n=1, 2...k, SUC_n \times \ll generalization \gg \times GUC\}$ of *GUC*
F3.1 add an edge corresponding to $n_i \times SC_n \bigwedge_{m=1}^{n-1} NOT(SC_m) \bigwedge_{m=n+1}^{m=k} NOT(SC_m) \times n_c$ to *E*
F3.2**Add_Nodes_Corresponding_To_Steps** (n_c, $Steps_n$), such that n_s is the resulting sub-graph *end of main scenario* leaf or *failed scenario* leaf
F3.3 add an edge corresponding to $n_s \times null \times n_j$

Figure 6-1: Algorithm for generalization relationship transformation in the control flow-based state machine

In the case of Order Process System (abbr., OPSystem), a CFSM (Figure 6-3) is generated from a partial use case model of the OPSystem (Figure 6-2). The partial use case model is circled by a dash line. The main use case in this partial use case model is the use case: "Place Order" (detailed use case

descriptions can be found in Appendix I.). The CFSM takes into account the flow of events in its included use cases, extending use cases and specialized use cases. Edge *C1* is the extension condition leading to the extending use case: "Request Catalog". Edge *5* and *6* are the inclusion directives that lead to the included use case: "Update Account" and "Arrange Payment". Edge *6_2&SC1&NOT(SC2)* will lead to the use case: "Credit Card Payment" because its specialization condition (SC1) "Customer selects Credit Card Payment" is satisfied. Edge *6_2&SC2&NOT(SC1)* will lead to the use case: "Cheque Payment" because its specialization condition (SC2) "Customer selects Cheque Payment" is satisfied. The two null edges will lead to the general use case: "Arrange Payment" from its specialized use case respectively, and followed by the flows *6_3* and *6_4* in the general use case: "Arrange Payment". Thus, a complet e CFSM is created.

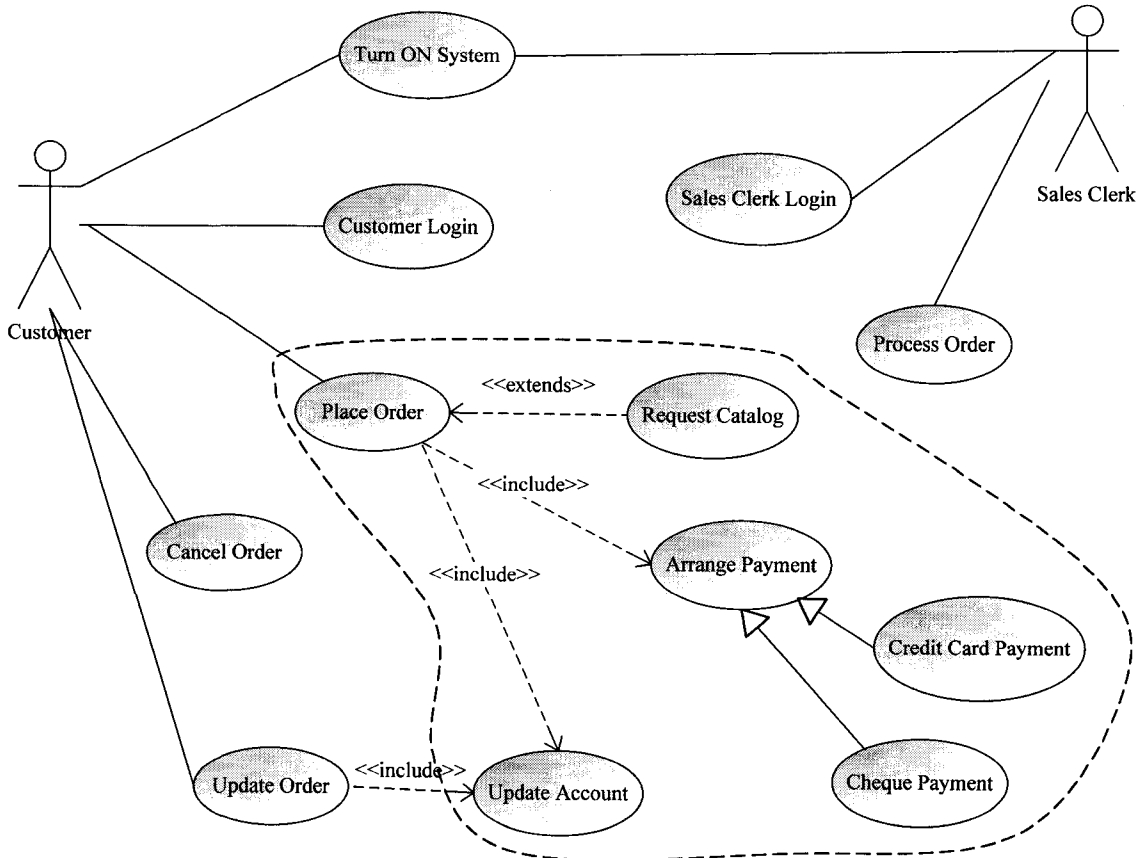


Figure 6-2: Use case model of the Order Process System. The partial use case model circled by a dash line is used to generate a control flow-based state machine.

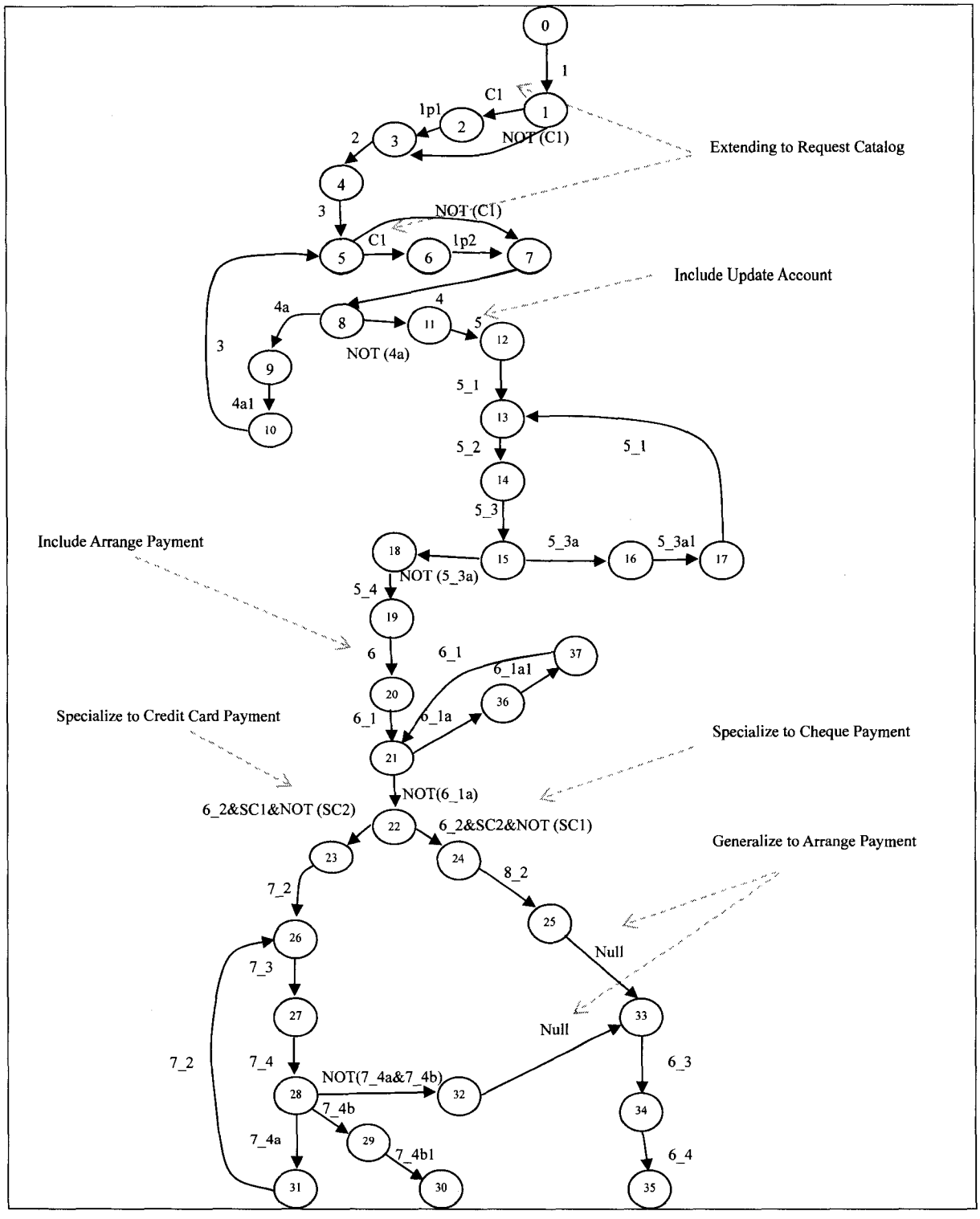


Figure 6-3: Control flow-based state machine generated from main use case: "Place Order", the flow of events from its auxiliary use cases: "Update Account", "Arrange Payment" (specialized use case: "Credit Card Payment" and "Cheque Payment") and "Request Catalog" are involved

6.2.2 Description of Path Sequence Extraction Algorithm

A path sequence is a sequence of nodes and edges $n_0 \xrightarrow{evl} n_1 \ldots \xrightarrow{evi} n_i$ starting from the root node of a CFSM. The CFSM generation relies on the control flows among different-level use cases (i.e., main use cases and their auxiliary use cases) and among same-level use cases (i.e., main use cases with implicit sequential relations) as specified by precede relations. The CFSM in Figure 6-3 is generated from the use case: "Place Order", which does not involve any succedent use cases. Figure 6-4 shows the CFSM generated from a precedent use case: "Customer Login" and two succedent use case: "Cancel Order" and "Update Order" to the use case: "Place Order". The precede relations are specified in the graph by dash lines. The use case: "Update Order" and "Cancel Order" can be executed simultaneously, which means their scenarios can be interwoven. However, in this thesis, we do not discuss interwoven scenarios, because too many combinations would be considered and it would cause a scenario explosion.

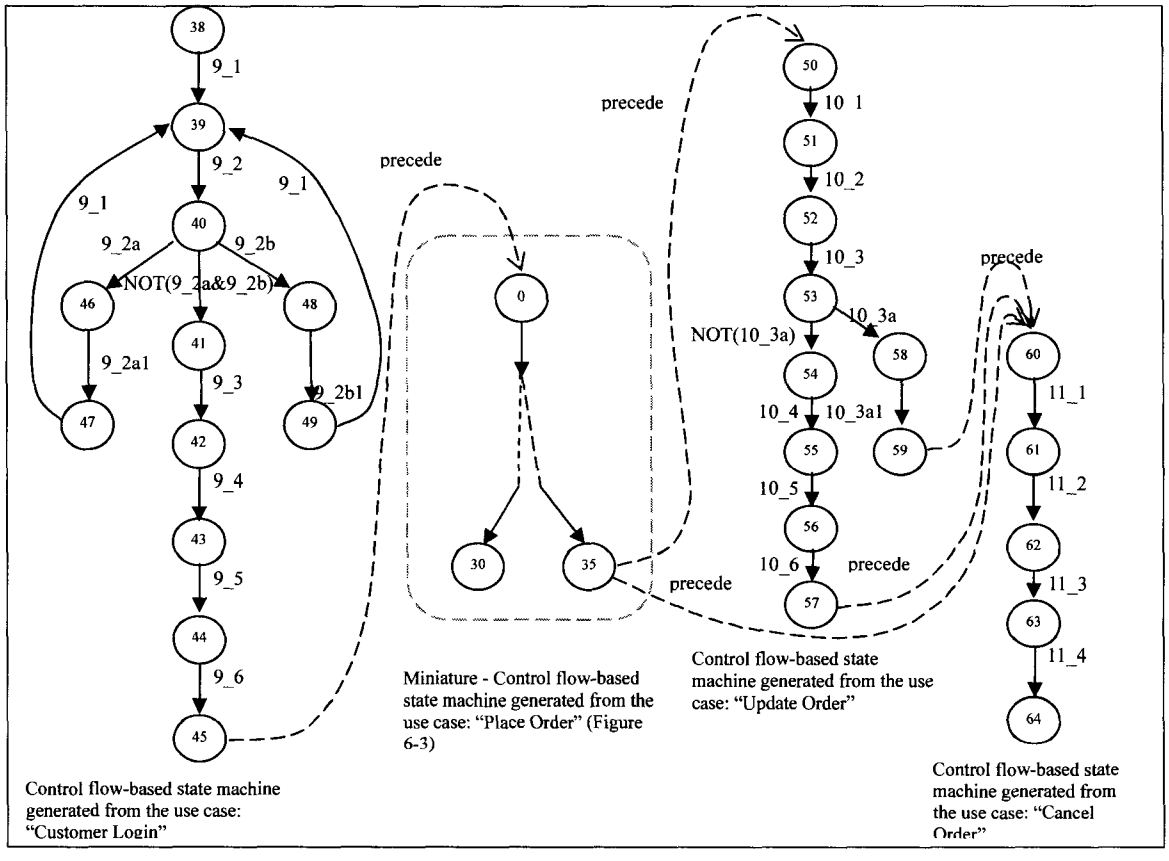


Figure 6-4: Control flow-based state machine generated from main use case: "Place Order", main use case: "Customer Login", main use case: "Update Order", main use case: "Cancel Order", they are connected by precede relations.

To extract path sequences from such a global CFSM, we employ depth-first traversal algorithm. The traversal algorithm starts building a path sequence from the root node (i.e., root node of the CFSM of the main use case), then it follows edges until it reaches a leaf node. The generated path sequences are a traversal set, which cover all control flows. In a path sequence, each single edge label corresponds to a condition or a step specified in use case descriptions. The difference here between path sequences and test scenarios is that the edge attributes (i.e., conditions or steps) are implicit in path sequences, while the attributes in test scenarios need to be explicit to testers. A solution for transforming path sequences to test scenarios will be provided in the following sections. A complete path sequence is activated by a primary actor, starting from the root node and ending in a leaf node. The path sequences that end at an alternative step are called alternative paths, and the path sequences that end at a main success scenario step are called main paths. The alternative paths and main paths constitute a complete path sequence set. Appendix II lists all path sequences extracted from the CFSM in Figure 6-3. In order to save space, we describe the path sequences with node number only, however, a complete path must involve the edge label between each node-pair as it provides critical information for generating test cases.

If two use cases have a precede relation, two CFSM are generated and a transition representing the precede relation will connect a certain leaf node of a CFSM (i.e., generated from the precedent use case) to the root node of another CFSM (i.e., generated from the succedent use case). If the depth-first traversal algorithm reaches the leaf node, it will continue traversing from the root node to obtain more edges to build a path sequence. For instance in Figure 6-4, in the CFSM of the use case: "Customer Login", after the step "OPSystem displays a homepage" (i.e., Edge *9_6*) is executed, the step "Customer selects items" (Edge *1*) in the CFSM of the use case: "Place Order" can be executed. We propose Definition 6-1 on the concatenation of two path sequences. Based on Definition 6-1, we propose Definition 6-2 on the concatenation of path sequences of two use cases. If the precede relations are captured from success post-conditions and pre-conditions, the concatenation of path sequences will be conducted between the main paths of the precedence use case and all paths of the succedent use case. If the precede relations are captured from alternative post-conditions and pre-conditions, the concatenation of path sequences will be conducted between the alternative paths which lead to the alternative post-conditions and all paths of the succedent use case. For instance in Figure 6-5, a sample path-sequence concatenation is shown. The path sequences are generated from the CFSM presented in Figure 6-4. Since there are *80* path sequences

generated from the CFSM of the main use case: "Place Order", we only choose one main path from them.

Definition 6-1: Concatenation of two path sequences

The concatenation of two path sequences is that,

$p_i = n_0 \xrightarrow{ev_0} n_1 \ldots n_{i-1} \xrightarrow{ev_{i-1}} n_i$

$p_s = n_i \xrightarrow{ev_i} \ldots n_k \xrightarrow{ev_k} n_k$,

such that path sequence $p = p_i + p_s$, $p = n_0 \xrightarrow{ev_0} n_1 \ldots n_{i-1} \xrightarrow{ev_{i-1}} n_i \xrightarrow{ev_i} \ldots n_k \xrightarrow{ev_k} n_k$

Definition 6-2: Concatenation of path sequences from two use cases

- Given two use cases UC_1 and UC_2, IF $post(UC_1) \Rightarrow pre(UC_2)$,

The concatenation of UC_1 and UC_2 path sequences is a set of path sequence *ConcPath* such that:

$ConcPath = \{p \mid p = n_0 \xrightarrow{ev_1} n_1 \ldots \xrightarrow{ev_i} n_i \xrightarrow{ev_{i+1}} n_{i+1} \ldots n_{j-1} \xrightarrow{ev_j} n_j\}$ with

$\forall p_i, p_i = n_0 \xrightarrow{ev_1} n_1 \ldots \xrightarrow{ev_i} n_i \in MainPaths(UC_1)$

$\forall p_j, p_j = n_i \xrightarrow{ev_{i+1}} n_{i+1} \ldots n_{j-1} \xrightarrow{ev_j} n_j \in AllPaths(UC_2)$,

- Given two use cases UC_1 and UC_2, IF $\exists Altpost \in Altpost(UC_1) \mid Altpost \Rightarrow pre(UC_2)$,

The concatenation of UC_1 and UC_2 path sequences is a set of path sequence *ConcPath* such that:

$ConcPath = \{p \mid p = n_0 \xrightarrow{ev_1} n_1 \ldots \xrightarrow{ev_i} n_i \xrightarrow{ev_{i+1}} n_{i+1} \ldots n_{j-1} \xrightarrow{ev_j} n_j\}$ with

$\exists p_i, p_i = n_0 \xrightarrow{ev_1} n_1 \ldots \xrightarrow{ev_i} n_i \in AltPaths(UC_1)$,

$\forall p_j, p_j = n_i \xrightarrow{ev_{i+1}} n_{i+1} \ldots n_{j-1} \xrightarrow{ev_j} n_j \in AllPaths(UC_2)$,

Under the assumption that the interwoven scenarios are not taken into account, the number of the path sequences can then be obtained. We assume that the number of all paths generated from the CFSM of the use case UC_2 is k, the number of main paths generated from the CFSM of the use case UC_1 is m and the number of alternative paths is n. Then, iff $post(UC_1) \Rightarrow pre(UC_2)$, the number of the concatenated path sequences is $m \times k + n$. If there exists an alternative post-condition of UC_1 such that iff $Altpost(UC_1) \Rightarrow pre(UC_2)$, the number of the alternative paths that lead to this alternative post-condition is n_1, then the number of the concatenated path sequences is $n_1 \times k + m$. For instance, *2* path sequences are generated from the main use case: "Update Order", in which one is a main path sequence and the other one is an alternative path sequence. One path sequence is generated from the use case: "Cancel Order". Therefore, 2 $(1 \times 1 + 1)$ concatenated path sequences are generated. As a more complicated example, in the *80* path sequences generated from the CFSM of the use case: "Place Order", there are *32* alternative paths and *48* main paths. Therefore, there

will be $(32+48\times2+48\times1) = 176$ path sequences being generated from the CFSM of the main use case: "Place Order" with two succedent use cases: "Update Order" and "Cancel order".

From Use case Customer Login:
38-39-40-41-42-43-44-45
38-39-40-46-47-39-40-41-42-43-44-45
38-39-40-46-47-39-40-48-49-39-40-41-42-43-44-45
38-39-40-48-49-39-40-46-47-39-40-41-42-43-44-45
} Main paths

From Use case Place Order (choose one):
0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35 } Main path

From Use case Update Order:
50-51-52-53-58-59 } Alternative path 50-51-52-53-54-55-56-57 } Main path

From Use case Cancel Order
60-61-62-63-64 } Main path

The concatenation will be

- 38-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-50-51-52-53-58-59-60-61-62-63-64
- 38-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-50-51-52-53-54-55-56-57-60-61-62-63-64
- 38-39-40-46-47-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-50-51-52-53-58-59-60-61-62-63-64
- 38-39-40-46-47-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-50-51-52-53-54-55-56-57-60-61-62-63-64
- 38-39-40-46-47-39-40-48-49-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-50-51-52-53-58-59-60-61-62-63-64
- 38-39-40-46-47-39-40-48-49-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-50-51-52-53-54-55-56-57-60-61-62-63-64
- 38-39-40-48-49-39-40-46-47-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-50-51-52-53-58-59-60-61-62-63-64
- 38-39-40-48-49-39-40-46-47-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-50-51-52-53-54-55-56-57-60-61-62-63-64
- 38-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-60-61-62-63-64
- 38-39-40-46-47-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-60-61-62-63-64
- 38-39-40-46-47-39-40-48-49-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-60-61-62-63-64
- 38-39-40-48-49-39-40-46-47-39-40-41-42-43-44-45-*0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35*-60-61-62-63-64

Figure 6-5: Sample path sequence concatenation from the control flow-based state machine presented in Figure 6-4

6.3 Algorithm of Path Sequence Generation

As noted, many path sequences are generated. In order to decrease the number of test scenarios while keeping the adequacy of testing, test coverage criteria should be applied based on path sequences. Thus, test cases inferred from test scenarios can be generated for effective system testing. Test coverage is a quality measure of testing. Test coverage is of great importance when testing because testers may not have enough time to execute all test cases. Thus, test coverage can guide the testers to the areas that require testing and give a measure of confidence. This idea is very practical in real world and it is essential to maximize the quality of product development while minimize time spent. Recall that in Chapter 2, we reviewed six coverage criteria based on a code excerpt. The difference between the reviewed control-flow graph and our CFSM is that the node in CFSM does not include critical information for path sequences. However, in the control-flow graph presented in Chapter 2, the nodes include information such as conditions and statements.

In this section, we propose detail algorithm for three test coverage criteria so that the generated path sequence will be adequate. Another objective of applying test coverage in our approach is to demonstrate that the generation process can be automated. There are a variety of test coverage criteria such as data test coverage, non-functional test coverage proposed in SCENT [Rys99], decision-table test coverage in TOTEM [Bri02]. Automating those test coverage criteria will be considered as our future research. In the following sections, path sequences will be used to infer test scenarios, which will be further used to create test cases. This process will be realized by implementation as a tool. Test case creation is manual, and it will be considered as another aspect of our future research.

6.3.1 Complete-Path Test Coverage Algorithm

In section 2.8.6, we reviewed that complete-path coverage is the strongest criterion and it requires that all paths be exercised. One difference between our CFSM and the reviewed control-flow graph in Chapter 2 is that the loop in the reviewed control-flow graph terminates on a certain condition. However, in the CFSM, the number to visit a loop is determined by the test requirements, which means a limit is required, or the loop will be executed indefinitely. The loops may be regarded as recovery scenarios (i.e., GOTO Steps). Therefore, in our complete path coverage on traversing

CFSMs, the path sequences are generated according to a repetition limit, *l*, such that a loop does not appear in a path more than *l* times. With this repetition limit in mind, infinite loops can be avoided.

Generate_Complete_path_sequence (*g*: CFSM, *l*: integer) *returns* a set of path sequences

G1. *PathResults* = Ø, *path* = Ø, let *R* be the root node of graph *g*

G2. **Traverse** (*R, l, path, PathResults*)

G3. return *PathResults*

Traverse (*n*: node, *l*: integer, *p*: path, *PathResults*: set of paths)

H1. IF *n* is a *leaf* node,

H1.1 *PathResults* = *PathResults* $\cup$ {*p*}

H2. FOR EACH edge $n \xrightarrow{e} n_i$ starting from *n*

H2.1 IF node n_i doesn't appear in *p* at least *l+1* times

H2.1.1 p = p $\xrightarrow{e}$ n_i

H2.1.2 **Traverse** (n_i, *l, p, All*)

Figure 6-6: Core algorithm of generating path sequences from system-level control flow-based state machine, and with Complete-path test coverage criterion

The whole process of path sequence generation according to the complete-path test coverage criterion is described in the algorithm (Figure 6-6). The traverse is based on the control flow-based state machines that are enhanced by sequential relations and takes into account the repetition limit *l*. When a new path is built, it will be added to "*PathResults*" (H1.1). If the algorithm does not reach the leaf, it will recursively be executed while adding edge to build a path sequence (H2.1.1). If a node *n* does not appear in path *p* for at least (*l+1*) times, the outgoing edge of *n* can be added to the path *p*. Let us illustrate this algorithm by using a simple example. Figure 6-7 is a simple CFSM. If the repetition limit is set to *1*, after the node *2* and node *1* have been visited for the first time, they can be visited a second time, thus the loop *2-1* can only appear once. If the repetition limit is set to *2*, after node *2* and node *1* have been visited for a second time, they can be visited a third time, thus the loop *2-1* can appear twice. For the CFSM in Figure 6-3, the *80* path sequences generated in Appendix II according to complete-path test coverage criterion and the repetition limit is *1*. If the repetition limit is set up more than *1*, more path sequences will be generated.

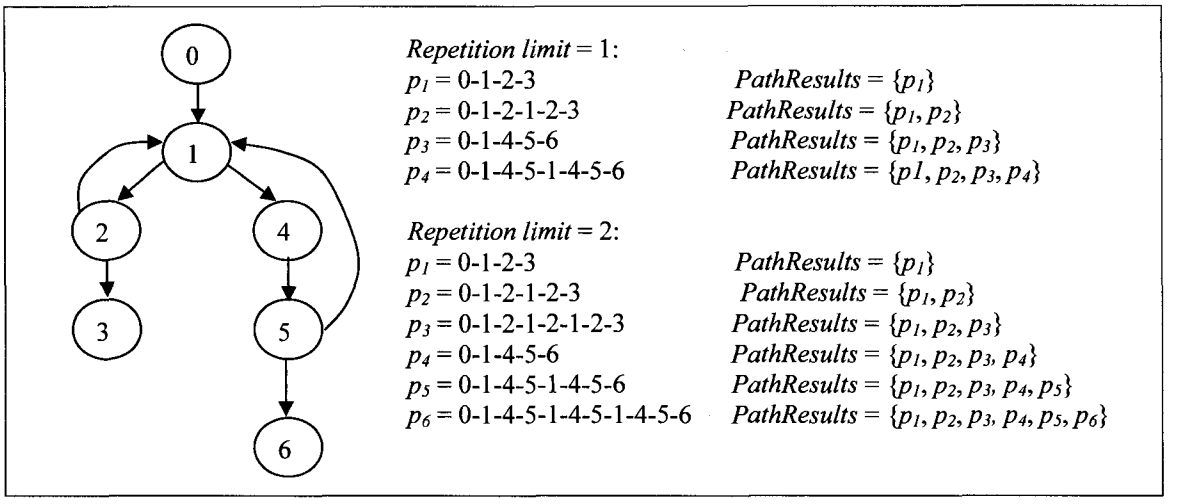


Figure 6-7: Example to illustrate the path sequence generation according to complete-path coverage

6.3.2 All-Node Test Coverage Algorithm

All-node test coverage achieves a set of path sequence such that each node in a CFSM appears at least once. Figure 6-8 presents the core algorithm to generate all-node test-coverage path sequences. The "*AllVisitedNodes*" is a temporary set to store the nodes, which have been visited when building a path sequence. If the traversal does not reach a leaf, it will recursively be executed while adding edges that destination nodes of the edges are unvisited nodes to build a path (J4.1). We find that due to the existence of loops, the destination node of an edge that represents a loop will never be visited twice. Therefore, the path sequences satisfying the all-node test coverage will not cover all loops in the CFSM. This brings another limitation, an uncompleted path will occur when generating path sequences. For instance, in the CFSM in Figure 6-3, when the traversal reaches the node *31*, it will not visit node *26* such that an uncompleted path is built and the traversal cannot continue. In addition, these uncompleted paths are not sufficient to generate complete test cases. In order to solve this problem, these uncompleted paths must be augmented to complete paths. When a new path sequence is built (i.e., reach a leaf), it will be added to a set called "*AllCompletePaths*" (J2.1). This set stores all complete paths. Accordingly, another set called "*AllPartialPaths*" is required. When the traversal reaches a node *n* such that the destination nodes of all outgoing edges of node *n* have been visited and a complete path is not built, the partial path will be added in "*AllPartialPaths*" (J3.1). A function called **Complete_Partial_Paths** (*AllCompletePaths*, *AllPartialPaths*) is required to complete the paths in "*AllPartialPaths*". For each path in "*AllPartialPaths*", the algorithm will search every path in "*AllCompletePaths*" (K1.3). If a sub-path starts from the node *n*, which is the last node of the

current partial path, and the node *n* only appears once in the sub-path (i.e., keep the minimum length so as to make the algorithm more effective), it will be stored in the set "*TempPaths*". After all paths in "*AllCompletePaths*" have been searched, the path with minimum length will be concatenated to the current partial path in order to build a complete path. Thus, all partial paths in "*AllPartialPaths*" are completed and can be used to infer concrete test cases.

Generate_All_Node_path_sequence (*g*: CFSM) *returns* a set of path sequences
I1 $p = \emptyset$, *AllVisitedNodes* = $\emptyset$, *AllCompletePaths* = $\emptyset$, *AllPartialPaths* = $\emptyset$
  let *R* be the root node of graph g;
I2 **Traverse_for_All_Node_Coverage** (*R*, *p*, *AllVisitedNodes*, *AllCompletePaths*, *AllPartialPaths*);
I3 **Complete_Partial_Paths** (*AllCompletePaths*, *AllPartialPaths*);
I4 return *AllCompletePaths*;

Traverse_for_All_Node_Coverage (*n*: node, *p*: path, *AllVisitedNodes*: a set of nodes, *AllCompletePaths*, *AllPartialPaths*: a set of paths)
J1 *AllVisitedNodes* = *AllVisitedNodes* $\cup \{n\}$;
J2 IF *n* is a *leaf* node
 J2.1 *AllCompletePaths* = *AllCompletePaths* $\cup \{p\}$;
J3 ELSE IF FOR all outgoing edges $n \xrightarrow{e} n_i$ from *n*, $n_i \in$ *AllVisitedNodes*
 J3.1 *AllPartialPaths* = *AllPartialPaths* $\cup \{p \xrightarrow{e} n_i\}$;
J4 ELSE FOR EACH edge $n \xrightarrow{e} n_i$ starting from *n*
 J4.1 IF $n_i \notin$ *AllVisitedNodes*
  J4.1.1 $p = p \xrightarrow{e} n_i$;
  J4.1.2 **Traverse_for_All_Node_Coverage** (n_i, *p*, *AllVisitedNodes*) ;

Complete_Partial_Paths (*AllCompletePaths*, *AllPartialPaths*);
K1 For each $p_i \in$ *AllPartialPaths*;
 K1.1 Let n_i be the last node of p_i;
 K1.2 *TempPath* = $\emptyset$;
 K1.3 For each $p_j \in$ *AllCompletePaths*;
  K1.3.1 Let n_j be the last node of p_j;
  K1.3.2 Find sub-path p_s starting n_i to n_j, n_i appear only once;
  K1.3.3 *TempPaths* = *TempPaths* $\cup \{ p_s \}$
 K1.4 For each $p_s \in$ *TempPaths*
  K1.4.1 Find p_s which has the minimum length
  K1.4.2 $p_i = p_i + p_s$;
  K1.4.3 *AllCompletePaths* = *AllCompletePaths* $\cup \{p_i\}$;

Figure 6-8: algorithm of generating path sequences from system-level control flow-based state machine with all-node test coverage

Let us illustrate this algorithm by using a simple example. Figure 6-9 is a simple CFSM. When a complete path sequence p_1 is built and the traversal back traces to node *2*, it will not visit node *1* because node *1* has already been visited. Thus, a partial path p_2 is added in "*AllPartialPaths*". When a

complete path sequence p_3 is built and the traversal back trace to node *5*, it will visit node *7* but will not visit node *1* again because it has already been visited. Thus, another partial path sequence p_4 is added to "*AllPartialPaths*". For the two partial paths in "*AllPartialPaths*", a sub-path 1-2-3 with minimum length is found, it will be concatenated to p_2 and p_4 to make them complete.

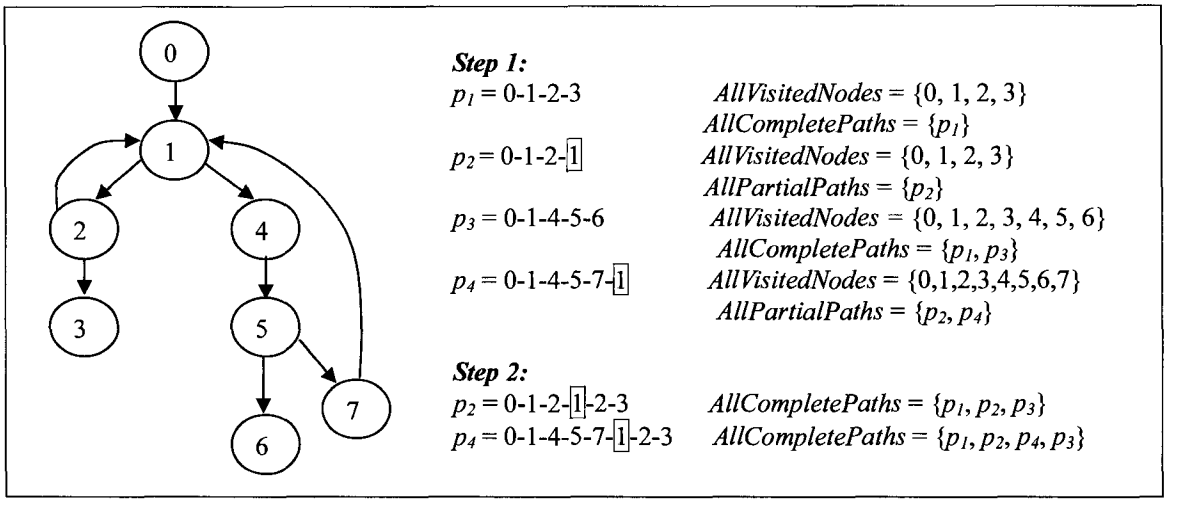


Figure 6-9: Example to illustrate the path sequence generation according to all-node coverage

For the CFSM in Figure 6-3 generated from the partial OPSystem use case model, *2* complete paths and *5* partial paths will be generated first according to all-node coverage algorithm. Then, a sub-path will be found from one of the complete paths to concatenate to each partial path. For each partial path, the first node of the sub-path must be the same as the last node (i.e., the node with a square) of the partial path and the length of the sub-path should have minimum length. Below are the generated complete paths and partial paths:

- **Complete paths:**

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-29-30

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35

- **Partial paths:**

0-1-2-3-4-5-6-7-8-9-10-[5]

0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-[13]

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-[21]

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-[26]

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-24-25-[33]

- **Complete paths after completing partial paths:**

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28- 29-30

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35

0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28- 29-30

0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28- 29-30

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28- 29-30

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28- 29-30

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35

6.3.3 All-Edge Test Coverage Algorithm

All-edge test coverage achieves a set of path sequences in which each edge in the CFSM appears at least once. Figure 6-10 presents the core algorithm for extracting a set of path sequences satisfying the all-edge test coverage criterion by traversing the CFSM. The "*AllVisitedEdges*" is a temporary set to store the edges, which have been visited when building a path sequence. If the traversal does not reach a leaf, it will recursively be executed while adding unvisited edges to build a path (M3.1). When a new path sequence is built (i.e., reach a leaf), it will be added into a set called "*AllCompletePaths*" (M1.1). This set stores all complete paths. Accordingly, another set called "*AllPartialPaths*" is required. When the traversal back traces to a node *n*, such that all outgoing edges of node *n* have been visited and a complete path has not been built, it will be added to "*AllPartialPaths*" (M2). When the traversal ends, the paths in "*AllCompletePaths*" and "*AllPartialPaths*" satisfy the all-edge coverage. However, a test case inferred from such paths in "*AllPartialPaths*" is not complete. Therefore, a function **Complete_Partial_Paths** (*AllCompletePaths*, *AllPartialPaths*) is required to complete the paths in "*AllPartialPaths*". For each path in "*AllPartialPaths*", the algorithm will search every path in "*AllCompletePaths*". If a sub-path starts from node *n*, which is the last node of the current partial path, and node *n* only appears once in the sub-path (i.e., keep the minimum length so as to make the algorithm more effective), it will be stored in the set "*TempPaths*". After all paths in "*AllCompletePaths*" have been searched, the path with minimum length will be concatenated to the current partial path in order to build a complete path. Thus, all partial paths in "*AllPartialPaths*" are completed and can be used to infer complete test cases.

Generate_All_Edge_path_sequence (*g*: CFSM) *returns* a set of path sequences

L1 *AllCompletePaths* = Ø; *AllPartialPaths* = Ø; *AllVisitedEdges* = Ø; *p* = Ø,

let *R* be the root node of graph *g*;

L2 **Traverse_for_All_Edge_Coverage** (*R*, *p*, *AllVisitedEdges*, *AllCompletePaths*, *AllPartialPaths*);

L3 **Complete_Partial_Paths** (*AllCompletePaths*, *AllPartialPaths*);

L4 return *AllCompletePaths*;

Traverse_for_All_Edge_Coverage (*n*: node, *p*: path, *AllVisitedEdges*: a set of edges, *AllCompletePaths*, *AllPartialPaths*: a set of paths)

M1 IF *n* is a *leaf* node

M1.1 *AllCompletePaths* = *AllCompletePaths* $\cup \{p\}$;

M2 ELSE IF FOR all outgoing edges $n \xrightarrow{e} n_i$ from *n*, $n \xrightarrow{e} n_i \in AllVisitedEdges$

M2.1 *AllPartialPaths* = *AllPartialPaths* $\cup \{p\}$;

M3 ELSE FOR EACH edge $n \xrightarrow{e} n_i$ starting from *n*

M3.1 IF $n \xrightarrow{e} n_i \notin AllVisitedEdges$;

M3.1.1 $p = p \xrightarrow{e} n_i$; $AllVisitedEdges = AllVisitedEdges \cup \{ n \xrightarrow{e} n_i \}$;

M3.1.2 **Traverse_for_All_Edge_Coverage** (n_i, *p*, *AllVisitedEdges*) ;

Figure 6-10: Core algorithm of generating path sequences from system level control flow-based state machine, and with all-edge test coverage criterion

Let us illustrate this algorithm by using a simple example. Figure 6-11 is a simple CFSM. When a complete path sequence p_1 is built and the traversal back traces to node *2*, it will visit edge *2-1* but will not visit edge *1-2* because it has already been visited. Thus, edge *1-4* will be visited and another complete path, p_2, is built. When the traversal back traces to node *5* and visit edge *5-7* and *7-1*, all outgoing edges of node *1* have been visited, thus, a partial path, p_3, is built. Then, the two sub-paths (i.e., *1-2-3*, *1-4-5-6*) can be found from the complete paths. The sub-path with minimum length (i.e., *1-2-3*) will be concatenated to the partial path p_3. Thus, p_3 is completed (i.e., *0-1-2-1-4-5-7-1-2-3*).

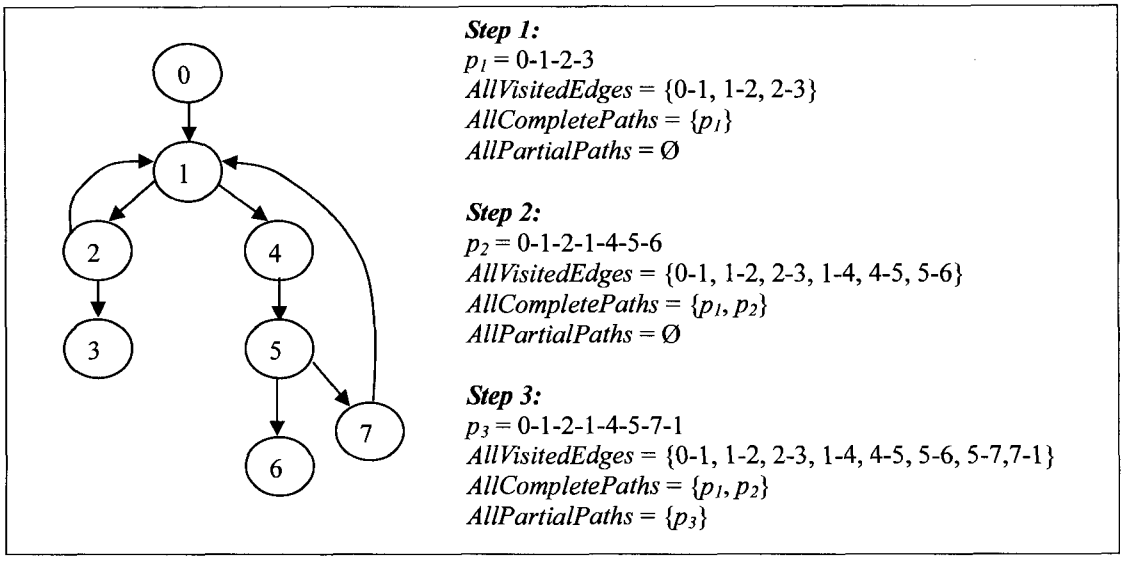


Figure 6-11: Example to illustrate the path sequence generation according to all-edge coverage

For the CFSM in Figure 6-3, generated from the partial OPSystem use case model, *2* complete paths and *7* partial paths will be generated first according to the all-edge test coverage algorithm. Within the *7* partial paths, two of them require manual correction. They are indicated by (*). In each single path sequence, if an edge representing an extension condition appears more than once, the extension condition should evaluate to the same value, i.e., both evaluate to true or both evaluate to false. For instance, in the CFSM in Figure 6-3, the edge *C1* represents an extension condition "Catalog Requested". If "C1" has appeared in a path sequence, then "NOT(C1)" cannot appear and vice versa. Therefore, the partial path "0-1-2-3-4-5-6-7-8-9-10-5-7" should be changed to "0-1-2-3-4-5-6-7-8-9-10-5" as the traversal cannot visit the edge *5-7* (i.e., NOT (C1)). The partial path "0-1-3" is not useful as the sub-path in the complete paths has already visited edge *1-2* (i.e., C1), which is in conflict with edge *1-3* (i.e., NOT (C1)). However, according to the feature of all-edge coverage, the eventually generated *7* complete paths do not cover the edge *1-3* and *5-7*. Therefore, *7* new complete paths will be converted from the generated *7* complete paths by changing the value of the extension conditions to the opposite. This might be a limitation in the depth-first traversal, as it will never know that condition which has been visited, hence it can only be done manually so far.

- **Complete paths:**

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-29-30

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35

- **Partial paths:**

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-24-25-33

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21

0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13

0-1-2-3-4-5-6-7-8-9-10-5-7 (*)

0-1-3 (*)

- **Complete paths after completing partial paths:**

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-29-30

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-29-30

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-24-25-33-34-35

0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-29-30

0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-29-30

0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-29-30

- **New converted complete paths by changing the extension condition. :**

0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-29-30

0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35

0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-29-30

0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-22-24-25-33-34-35

0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-29-30

0-1-3-4-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-29-30

0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-29-30

6.4 Implementation

6.4.1 Objective

Previously in section 2.2.2, we introduced that a scenario is an execution path that describes the interactions between an actor and a system. Scenarios can be used to analyze a software system and test a system. One of the roles of the scenarios is to provide system testing. Scenarios lead to a view that is more transactional. They are apt to test flows of actions, from triggering events to results

[Rys99]. In Chapter 4, we introduced that in UCEd, scenarios are manually written. In addition, there are two ways to compose the scenarios: (i) Input arbitrarily in the UCEd scenario-editing tool; (ii) Adopt the simulation results from UCEd simulator in the UCEd scenario-editing tool. However, both ways are time-consuming, and there is no indication to the user that the necessary scenarios have been generated for testing. In addition, the adequacy criterion should be applied, yet the arbitrary way is not proper and does not conform to the criteria. Test cases can be directly derived from test scenarios. In previous sections, a path sequence generation approach was introduced. These generated path sequences only have nodes and edges, which are required to make the attribute of each edge explicit. In this section, test scenarios are derived from the generated path sequences.

The objective of implementation is to provide a flexible solution to overcome the time-consuming problem. This solution is flexible, because the user can generate a different set of system-level path sequences from different main use cases. Another advantage is that the scenario viewer will provide explicit vision of each edge in the generated path sequences, thus avoid determining what attribute each single edge has. A transformation from path sequences attribute to test scenarios attribute will be presented in the following sections, hence, the test scenarios are formed. These test scenarios can be edited in the scenario-editing tool.

6.4.2 Interface and Functional Design

In this section, a tool supporting automated system-level test-scenario generation is built. The tool starts with selecting one main use case. Path sequences are then generated from the selected use case. Users can select one of the three test coverage from the tool to generate corresponding reduced path sequences. Lastly, test scenarios are stored in a scenario model and the users can edit them in the scenario-editing tool. Briefly, the functions of the tool can be categorized into two parts.

1. *Generate path sequences satisfying a certain test coverage criterion corresponding to selected use cases.*
2. *Transform path sequences to test scenarios and then store test scenarios in the scenario model.*

In this section, we first introduce the interface of the path sequence generation. Then, we introduce the detail design of transformation from path sequences to test scenarios. Although the implementation is based on UCEd, the approach presented in previous sections, as well as the

implementation, can be applied in other situations with a little modification.

One limitation of current UCEd release is that it does not support generation relationships. In order to implement our approach, the replacement of abstract steps to concrete steps must be done manually when writing use case descriptions. The steps in a general use case will be moved to the specialized use cases associated with the general use case. For instance, the steps in the use case: "Arrange Payment" will be moved to its specialized use case: "Credit Card Payment" and "Cheque Payment" respectively. Then, when generating path sequences, only one specialized use case will be taken into account at a time. Therefore, the use case: "Place Order" will include either the use case: "Cheque Payment" or the use case: "Credit Card Payment".

6.4.2.1 Path Sequence Generation Interface

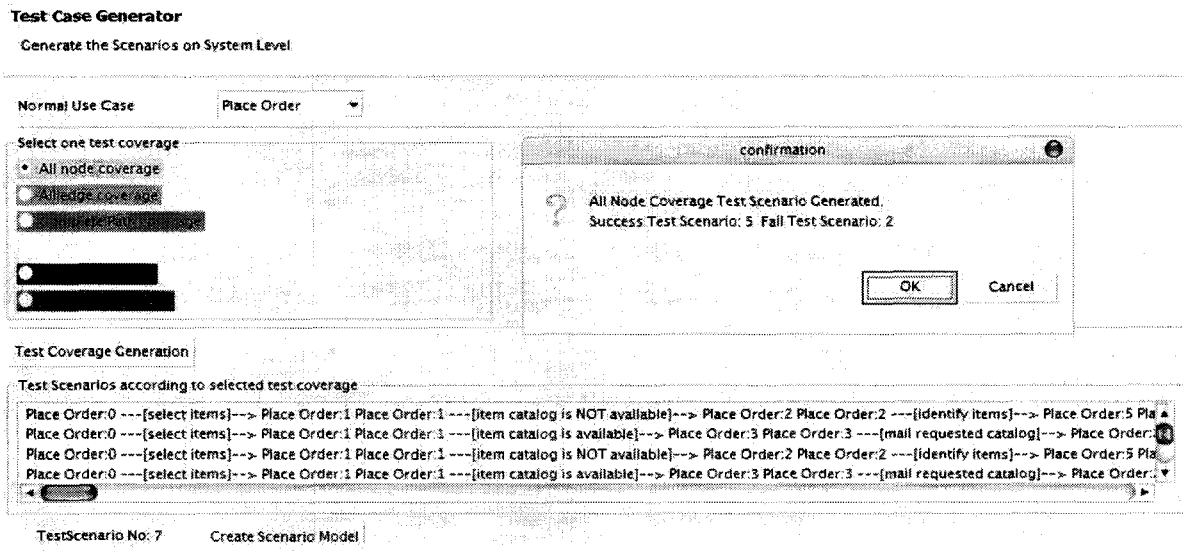


Figure 6-12: All-node test coverage from control flow-based state machine generated from the use case: "Place Order" with three succedent use cases: "Cancel Order", "Update Order" and "Process Order".

The interface of the tool complies with the path-sequence generation process. Users can choose a main use case and then generate path sequences according to a certain test coverage criterion. Figure 6-12 is a view of path sequences generated from the use case: "Place Order" with three succedent use cases (the use cases: "Cancel Order" and the use case: "Update Order" and the use case: "Process Order") based on the all-node test coverage criterion. There are *7* path sequences generated, in which *5* are main paths and *2* are alternative paths. Recall that in section 6.2.2, we

introduced that the system-level main paths and alternative paths come from the last use case (if more than one use case) that the path sequences traverse. In the *5* main paths of the path sequence results, three of them come from achieving the main goal of the use case: "Update Order", one of them comes from achieving the main goal of the use case: "Cancel Order" and one of them comes from achieving the main goal of the use case: "Process Order". In the *2* alternative paths of the path sequence results, one of them comes from the failure of the use case: "Update Order" and one of them come from the failure of the use case: "Place Order". The detail content of the path sequences are listed in Appendix III representing as the form of test scenarios. The approach of transforming path sequences to test scenarios will be introduced afterwards.

6.4.2.2 Automated Test Scenario Simulation

In this section, we use a scenario model as a container to store test scenarios. These test scenarios are automatically generated and then stored in the scenario model. Each test scenario is inferred from a single path sequence. As has been noted, each path sequence is a sequence of interactions. A test scenario can be extracted from a path sequence to *triggers*, *system reactions*, *waiting delays*, *guard realizations* and *assertions*. Recall from section 4.7, an *assertion* must be valid according to the system's state when evaluated, *triggers* are provided as input events to the system, *system reactions* specify operations that need to be produced by the system and *guards* are conditions that must be enabled for a test scenario to proceed. A test scenario execution fails when an *assertion* is not verified or the system does not produce a specified reaction. Figure 6-13 presents the algorithm of deriving test scenarios from path sequences. The generated path sequences based on a certain test coverage criterion are mapped to test scenarios. Then, the test scenarios are stored in the corresponding scenario model according to the following steps:

1. Each scenario starts with a test condition corresponding to the use case pre-condition (O.1)
2. Each actor operation, system operation in a path sequence corresponds respectively to a trigger, a system reaction in the resulting test scenario (O.2.1).
3. Each condition in a path sequence corresponds respectively to a guard condition in the resulting test scenario (O.2.2). If the condition is a time out, it corresponds to a delay in the resulting test scenario (O.2.2.1).
4. A test scenario from an alternative path is completed with an assertion corresponding to the alternative post-condition (O.2.3.1).

5. A test scenario from a main path is completed with an assertion corresponding to the use case success post-condition (O.2.3.2).

N. **Create_Scenario_Model**(*PathResults*)
N.1 Create *ScenarioModel*;
N.1 **Add_Scenario**(*ScenarioModel*);

O. **Add_Scenario**(*ScenarioModel*)
O.1 Get *Pre-condition* from selected *use case*, create *Assertion*
Assertion ← *Pre-condition*
O.2 FOR EACH *path* in *PathResults*, FOR each *Edge* in *path*
O.2.1 IF*Edge* is *Operation*, get *Edge.Operation*
O.2.1.1 IF *Edge.Operation* is System Operation, create *SystemReaction*
SystemReaction ← *Edge.Operation*
O.2.1.2 IF *Edge.Operation* is Actor Operation, create *Trigger*
Trigger ← *Edge.Operation*
O.2.2 IF *Edge is Condition*, get *Edge.Condition*
O.2.2.1 IF *Edge.Condition* is Time out, create *Delay*
Delay ← *Edge.Condition*
O.2.2.2 ELSE create *Guard*
Guard ← *Edge.Condition*
O.2.3 IF *Edge*.Next = Ø, Get *Post-condition* from the *use case* inferred from the *Edge.*
O.2.3.1 IF this *Post-condition* is Alternative Post-condition, create *Assertion*
Assertion ← *use case* FAIL
O.2.3.2 IF this *Post-condition* is Success Post-condition, create *Assertion*
Assertion ← *use case* SUCCESS

Figure 6-13: Algorithm of Test Scenarios Inference from Path Sequences

The scenario viewer is a stage of scenario model, where the test scenarios are stored before being moved to the scenario-editing tool. Figure 6-14 presents a view of scenario viewer after we have created all-node coverage test scenarios and 7 (Appendix III) test scenarios satisfying the all-node test coverage are listed. In the right view, a test scenario is unwrapped and it is listed in Figure 6-15.

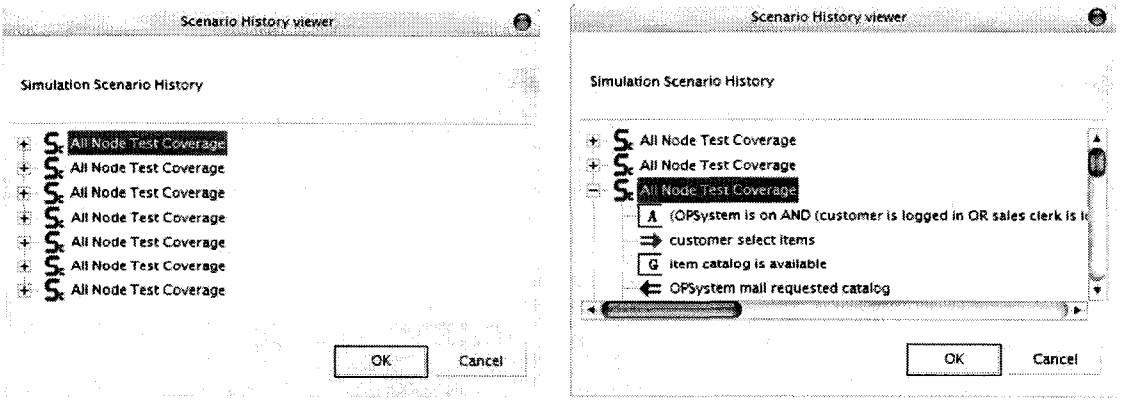


Figure 6-14: Scenario Viewer after we have created scenario model from path sequences, one of the test scenarios on the right view has been unwrapped

1. Assertion: (OPSystem is ON AND (Customer is logged in OR Sales Clerk is logged in))
2. Trigger: customer select items
3. Guard: Item catalog is available
4. System Reaction: OPSystem mail requested catalog
5. System Reaction: OPSystem identify items
6. Trigger: customer type item number
7. System Reaction: OPSystem validate item number
8. Guard: item number is invalid
9. System Reaction: OPSystem show item number invalid error message
10. Trigger: customer type item number
11. System Reaction: OPSystem validate item number
12. Guard: item number is not invalid
13. System Reaction: OPSystem display successful message
14. Trigger: customer account information is listed
15. Trigger: customer change customer account information
16. System Reaction: OPSystem verify customer account information
17. Guard: customer account information has typo
18. System Reaction: OPSystem shows account information error message
19. Trigger: customer change customer account information
20. System Reaction: OPSystem verify customer account information
21. Guard: customer account information has not typo
22. System Reaction: OPSystem display success message
23. Trigger: customer type credit card number
24. Trigger: customer type expiration date
25. System Reaction: OPSystem validate credit card information
26. Guard: Credit card information is valid
27. System Reaction: OPSystem display payment complete message
28. Trigger: customer requests OPSystem
29. System Reaction: OPSystem list placed orders
30. Trigger: customer change item number
31. Guard: item number is not invalid
32. Trigger: customer submit orders
33. System Reaction: OPSystem display success message
34. Trigger: customer account information is listed
35. Trigger: customer change customer account information
36. System Reaction: OPSystem verify customer account information
37. Guard: customer account information has typo
38. System Reaction: OPSystem shows account information error message
39. Trigger: customer change customer account information
40. System Reaction: OPSystem verify customer account information
41. Guard: customer account information has not typo
42. System Reaction: OPSystem display success message
43. Assertion: Update Order SUCCESS

Figure 6-15: One test scenario inferred from path sequence and stored in the scenario model

Line *1* is an assertion that also can be regarded as the test condition in the concrete test case. It is the pre-condition of the use case: "Place Order". Lines *2* to *13* are derived from the use case: "Place Order". Lines *14* to *22* are derived from the inclusion use case: "Update Account". Lines *23* to *27* are derived from the inclusion use case: "Credit Card Payment". Lines *28* to *33* are derived from the succedent use case: "Update Order" of the use case: "Place Order". Lines *34* to *42* are derived from the inclusion use case: "Update Account" of the use case: "Update Order". Finally, we have an assertion that the use case: "Update Order" has achieved its main success scenario.

6.5 Concrete Test Case Inference

In Chapter 5, we captured implicit use case sequential relations. System-level test scenarios can therefore be obtained. The test scenarios correspond to a certain test coverage criterion so that redundant test scenarios can be avoided. The entire process proceeds in a fully automated way. In this section, concrete test case generation from the test scenarios will be proposed. This stage can only be conducted manually so far, but the test scripts could be generated for other tools. It will be part of future research.

In section 2.8, we presented that a test case should contain a unique test case identifier, a test case name, test setup, inputs, guards and expected results and a test oracle. However, to generate concrete test cases, test data need to be added to the test inputs. For instance, Figure 6-16 is a concrete test case generated from the test scenario in Figure 6-15. The test case identifier and the test case name can be created manually. When building a test setup, the assertion in Line *1* is mapped to the test condition and the assertion in Line *43* is mapped to the test oracle. The test data assumption in the test setup is generated from every guard that the assumed test data value makes the guards true. Then, when testers execute test cases, the judgment of the guard comes from the test data in the test setup. The trigger is mapped to input with the test data in "<>" being manually added. System reaction is mapped to the expected results. Each concrete test case can be generated more than once from a test scenario with different test data.

Test data in the test setup can be derived from the guard conditions, which occur immediately following the test input. For instance, in the test setup, the "available item in catalog" is set up with the name of the items being X, Y, Z. These item names can make the guard "Item catalog is available" true (i.e., Line *2*). Another setup is for the valid item number which should be both less than *10* and positive. These item numbers can make the guard "Item number is not invalid" true (i.e., Line *11*), while making the guard "Item number is invalid" false (i.e., Line *7*). To sum up, the test data is divided according to invalid boundaries and valid boundaries. The sample invalid-boundary test data are mapped to the "False" guard condition and the sample valid-boundary test data are mapped to the "True" guard condition.

Test Case Identifier: ***I***
Test Case Name: ***Place Order precedes Update Order***
Test Setup:
- **Test Condition:** (OPSystem is ON AND (Customer is logged in OR Sales Clerk is logged in))
- **Test Oracle:** Update Order SUCCESS
- **Test Data Assumption**
  - Available item in catalog (Item name: X, Y, Z),
  - Valid item number (Item number: positive number less than 10)
  - Valid account information (Valid Account: not involve illegal letters such as $#@.)
  - Valid credit card number (Valid credit card number: 12 letters to 16 letters)
  - Valid expiration date (Valid expiration date: 1/1/2007 to 1/1/2010)
  - Valid customer attemps (1, 2, 3)
  - Valid delay time (less than 60 seconds)

1. Input: customer select items ***<Item Name>***
2. Guard: Item catalog is available
3. Expect Results OPSystem mail requested catalog
4. Expect Results: OPSystem identify items
5. Input: customer type item number ***<Number Value>***
6. Expect Results: OPSystem validate item number
7. Guard: item number is invalid
8. Expect Results: OPSystem show item number invalid error message
9. Input: customer type item number ***<Number Value>***
10. Expect Results: OPSystem validate item number
11. Guard: item number is not invalid
12. Expect Results:OPSystem display successful message
13. Input: customer account information is listed
14. Input: customer change customer account information ***<Account Information>***
15. Expect Results: OPSystem verify customer account information
16. Guard: customer account information has typo
17. Expect Results: OPSystem shows account information error message
18. Input: customer change customer account information ***<Account Information>***
19. Expect Results: OPSystem verify customer account information
20. Guard: customer account information has not typo
21. Expect Results: OPSystem display success message
22. Input: customer types credit card number ***<Credit Card Number>***
23. Input: customer types expiration date ***<Expiration Date>***
24. Expect Results: OPSystem validates credit card information
25. Guard: Credit card information is valid.
26. Expect Results: OPSystem display payment complete message
27. Input: customer requests OPSystem
28. Expect Results: OPSystem list placed orders
29. Input: customer change item number ***<Number Value>***
30. Guard: item number is not invalid
31. Input: customer submit orders
32. Expect Results: OPSystem display success message
33. Input: customer account information is listed
34. Input: customer change customer account information ***<Account Information>***
35. Expect Results: OPSystem verify customer account information
36. Guard: customer account information has typo
37. Expect Results: OPSystem shows account information error message
38. Input: customer change customer account information ***<Account Information>***
39. Expect Results:OPSystem verify customer account information
40. Guard: customer account information has not typo
41. Expect Results: OPSystem display success message

Figure 6-16: Test case inference from the test scenario in Figure 6-15

6.5.1 Possible Test Coverage on Concrete Test Case

Condition test coverage reports on the extent to which all possible outcomes are achieved for individual conditions composing a transition decision. By combining the test data into the test inputs, we are able to determine the outcome of each individual condition. For instance, all individual (atomic) conditions corresponding to the decisions from the CFSM in Figure 6-3 are listed in Figure 6-17. A sample combination of test data (Figure 6-18) can satisfy the condition coverage according to the test data assumption. Moreover, decision/condition coverage and condition combination coverage can be obtained.

Individual Condition	**C1** (*Catalog Requested*)	**4a** (*Item number is invalid*)	**5_3a** (*If customer account information has typo*)	**6_1a** (*After 60 seconds*)
	True (1-2, 5-6)	True (8-9)	True (15-16)	True (21-36)
	False (1-3, 5-7)	False (8-11)	False (15-18)	False (21-22)
Individual Condition	**7_4a_1** (*Credit card is invalid*)	**7_4a_2** (*Customer attempts <4*)	**7_4b_1** (*Credit card is invalid*)	**7_4b_2** (*Customer attempts =4*)
	True	True	True	True
	False	False	False	False

Figure 6-17: Atomic conditions corresponding to the condition

Test Data Assumption	TD1- Available item in catalog (Item name: X, Y, Z), TD2- Valid item number (Item number: positive number less than 10) TD3- Valid account information (Valid Account: not involve illegal letters such as $#@.) TD4- Valid credit card number (Valid credit card number: 12 letters to 16 letters) TD5- Valid expiration date (Valid expiration date: 1/1/2007 to 1/1/2010) TD6 - Valid customer attemps (1, 2, 3) TD7- Valid delay time (less than 60 seconds)							
Individual condition	**C1**	**4a**	**5_3a**	**6_1a**	**7_4a_1**	**7_4a_2**	**7_4b_1**	**7_4b_2**
TD1 = X TD2 = 9 TD3 = A032003 TD4=1030405020301982 TD5=09/30/2008 TD6= 3 TD7=30 seconds	T	F	F	F	F	T	F	F
TD1 = B TD2 = 11 TD3 = A03200$ TD4=10304050203019829 TD5=09/30/2008 TD6= 4 TD7=61 seconds	F	T	T	T	T	F	T	T

Figure 6-18: sample test data for condition test coverage

6.6 Chapter Summary and Highlights

In this chapter, we first introduced a method to create a complete CFSM from use case model considering generalization relationships. Then, an approach is presented to generate path sequences from the complete CFSM. Since the CFSM has been enhanced to a system level by using the sequential relations capture approach, the path sequences will integrate all the behaviours from the main use cases and their succedent use cases. Due to the high volume path sequences generated on a system level, test coverage should be applied to reduce the path sequence number while maintaining adequacy. In complete-path test coverage, we propose a repetition limit thus to avoid infinite loops. These reduced path sequences are mapped to test scenarios, which can be stored in a scenario model. The test scenarios can be edited in a scenario editing tool. The entire process is automated and it is implemented in UCEd. Lastly, concrete test cases are manually inferred from test scenarios.

Chapter 7 – CASE STUDY: *TEAM MANAGEMENT SYSTEM*

In this chapter, we analyze the approaches and tools presented in this thesis using a Team Management System (abbr., TMSystem). First, we introduce the TMSystem and its requirements. Then, we discuss the enhancement of the use case model using our sequential relation capture tool. Finally, we analyze the performance of the test scenario generation tool for a selected set of the use cases.

7.1 System Background

Students are typically asked to work on assignments and projects in teams as part of courses. Course instructors do not like to impose predetermined teams, and it can be difficult to set up these teams as students usually have a very short time to organize themselves. The students do not always know which other students are available. The collaboration between team members may also prove difficult as a project moves along because of lack of contact information. For instance, some students want to keep their phone and email information confidential from their teammates, but still need to be in contact. The TMSystem will help instructors and students set up and manage teams for assignments and projects. It runs as a client-server system. The server component shall run on an application server. Information about courses, students and instructors shall be processed from a database. Clients shall access the system using web browsers.

7.2 Requirements Description and Use Case Model

A high-level requirements description of the TMSystem is mentioned below. The TMSystem has three actors "Instructor", "Student" and "Liaison". The actor "Liaison" inherits from the actor "Student".

The system should allow the "Instructor" to:

- Modify properties set-up for teams anytime before a deadline (for teams' creation). The properties include a minimum number of students per team, a maximum number of students per team, deadline for the organization of teams and course section in which the students can be part of the teams. The new properties shall apply to all existing and future teams.
- Visualize a list of teams created for his course. For each team, there is a unique team identifier, a team name, a team date of creation, information about the team members with a mention of

the team liaison, and a status of a team as complete or incomplete (the minimum number of students is not reached). For each student as a team member, there is a student number, a student name, a student study program, the course section in which the student is enrolled.

- Add members to a team and allow to remove members from a team.

The system should allow "Student" to:

- Create teams once an instructor starts up the teams creation for a course. A team shall be created by specifying a name and a possible number of members.
- Automatically be a member of a team that he/she created.
- Automatically be a liaison for the team that he/she created.
- Visualize a list of all teams that are not complete.
- Ask to be added as a member of teams that are not complete.
- Quit teams in which they are members anytime before the deadline for teams' creation.
- Visualize the information about his team including the team identifier, the team name, the team date of creation, the team members' names with a mention of the team liaison and the status of a team as complete or incomplete.
- Visualize a list of names of students who asked to join his team

The system should allow "Liaison" to:

- Not allow a team to have less numbers than the minimum number. Not allow a team to have more members than the maximum number. Specify another team member as the team liaison.
- Add students from the list of students who asked to join that team as members of the team.

Additionally the TMSystem should:

- Prevent adding a student to a team with a status complete.
- Send an email notification to a student when he or she is added to a team or removed from a team.

A UML use case model for the TMSystem is represented in Figure 7-1. Use case descriptions of TMSystem will be presented in the next section.

Figure 7-1: Use Case Model of Team Management System

7.3 Use Case Description

7.3.1 Instructor's Use Cases

7.3.1.1 Use Case: Instructor Login

Title: Instructor log in
Scope: Design
Level: user
Primary Actor: Instructor
Goal: An instructor want to identify himself to the system in order to use it's functions
Pre-condition: TMSystem is ON AND Instructor is not logged in
Steps:
1. Instructor enters login information
2. TMS checks Instructor authorization status
3. TMS displays instructor operation choices
Alternatives:
2. a. Instructor is not authorized to use system
2. a. 1. TMSystem displays unauthorized access error message
Alternative Post-condition: TMSystem is ON AND Instructor is not logged in
Success Post-condition: TMSystem is ON AND Instructor is logged in AND TMSystem Display is instructor operation choices.

7.3.1.2 Use Case: Set up Parameters

Title: Set up parameters
Scope: Design
Level: user
Primary Actor: Instructor
Pre-condition: TMSystem is ON AND TMSystem Display is instructor operation choices AND Instructor is logged in
Steps:
1. Instructor chooses to set up teams parameters
2. TMSystem asks for teams parameters
3. Instructor provides parameters
4. TMSystem validates parameters
5. TMSystem displays teams start up acknowledgment message
6. AFTER 60 sec, TMSystem displays instructor operation choices
Alternatives:
4. a. Teams Parameters are not valid
4. a. 1. TMSystem displays teams parameters error message
4. a. 2. AFTER 60 sec, GOTO Step 2
Success Post-condition: TMSystem is ON AND Teams can be started up AND Instructor is logged in AND Parameter is set up AND TMSystem Display is instructor operation choices

7.3.1.3 Use Case: Modify Parameters

Title: Modify parameters
Scope: Design
Level: user
Primary Actor: Instructor
Pre-condition: TMSystem is ON AND TMSystem Display is instructor operation choices AND Instructor is logged in
Steps:
1. Instructor chooses to modify teams parameters
2. TMSystem asks for teams parameters
3. Instructor provides parameters
4. TMSystem validates parameters
5. TMSystem displays teams start up acknowledgment message
6. AFTER 60 sec, TMSystem displays instructor operation choices
Alternatives:
4. a. Teams Parameters are not valid
4. a. 1. TMSystem displays teams parameters error message
Alternative Post-condition: TMSystem is ON AND Instructor is logged in
Success Post-condition: TMSystem is ON AND Teams can be started up AND Instructor is logged in AND TMSystem Display is instructor operation choices.

7.3.1.4 Use Case: Visualize Student teams

Title: Visualize students teams
Scope: Design
Level: user
Primary Actor: Instructor

Pre-condition: TMSystem is ON AND TMSystem Display is instructor operation choices AND Instructor is logged in
Steps:
1. Instructor chooses to visualize students teams
2. TMSystem displays list of all teams
Success Post-condition: TMSystem is ON AND TMSystem Display is list of teams AND Instructor is logged in

7.3.1.5 Use Case: Modify a parameter for a team

Title: Modify a parameters for a team
Scope: Design
Level: user
Primary Actor: Instructor
Pre-condition: TMSystem is ON AND TMS Display is list of teams AND Instructor is logged in
Steps:
1. Instructor selects a team
2. TMSystem displays selected team information
3. Instructor choses team parameters modification
4. TMSystem displays team parameters
5. Instructor edits team parameters
6. TMSystem validates team parameters
7. TMSystem updates team parameters
8. AFTER 60 sec, TMSystem displays instructor operation choices
Alternatives:
6. a. Team parameters are not valid
6. a. 1. TMSystem displays teams parameters error message
6. a. 2. AFTER 60 sec GOTO 2
Success Post-condition: TMSystem is ON AND Teams parameters are changed AND Instructor is logged in AND TMSystem Display is instructor operation choices.

7.3.1.6 Use Case: Remove member from a team

Title: Remove member from a team
Scope: Design
Level: user
Primary Actor: Instructor
Pre-condition: TMSystem is ON AND TMS Display is list of teams AND Instructor is logged in
Steps:
1. Instructor selects a team
2. TMSystem displays selected team information
3. Instructor selects member for removal
4. TMSystem remove member from team
5. TMSystem sends notification to removed member
6. TMSystem displays instructor operation choices
Success Post-condition: TMSystem is ON AND Team has member removed AND Instructor is logged in AND TMSystem Display is instructor operation choices.

7.3.1.7 Use Case: Add member to a team

Title: Add member to a team
Scope: Design
Level: user
Primary Actor: Instructor
Pre-condition: TMSystem is ON AND TMSystem Display is list of teams AND Instructor is logged in
Steps:
1. Instructor selects a team
2. TMSystem displays selected team information
3. Instructor browses to selects student for addition from list of all students
4. TMSystem add student to the team
5. TMSystem sends notification to added member
6. TMSystem displays instructor operation choices
Alternative:
3. a. Student status is member of team
3. a. 1. TMS displays student already in team error message
3. a. 2. AFTER 60 sec GOTO 2
3. b. Team status is complete
3. b. 1. TMS displays team already complete error message
3. b. 2. AFTER 60 sec GOTO 2
Success Post-condition: TMSystem is ON AND Team has member added AND Instructor is logged in AND TMSystem Display is instructor operation choices.

7.3.1.8 Use Case: Instructor Logout

Title: Instructor log out
Scope: Design
Level: user
Primary Actor: Instructor
Pre-condition: Instructor is logged in AND TMSystem is ON
Steps:
1. Instructor logs out
2. TMSystem displays log out acknowledgment
Success Post-condition: Instructor is not logged in AND TMSystem is ON

7.3.2 Students' Use Cases

7.3.2.1 Use Case: Student Login

Title: Student log in
Scope: Design
Level: user
Primary Actor: Student
Goal: A Student want to identify himself to the system in order to use it's functions
Pre-condition: TMSystem is ON AND Student is not logged in
Steps:

1. Student enters login information
2. TMSystem checks Student authorization status
3. TMSystem displays student operation choices
Alternative:
2. a. Student is not authorized to use system
2. a. 1. TMSystem displays unauthorized access error message
Alternative Post-condition: Student is not logged in AND TMSystem is ON
Success Post-condition: Student is logged in AND TMSystem is ON AND TMSystem Display is student operation choices.

7.3.2.2 Use Case: Create Team

Title: Create team
Scope: Design
Level: user
Primary Actor: Student
Pre-condition: Student is logged in AND TMSystem Display is student operation choices AND TMSystem is ON
Steps:
1. Student selects new team creation operation
2. TMSystem asks for team name and students list
3. Student provides requested information
4. TMSystem validates provided information
5. TMSystem adds new team to teams
6. TMSystem displays student operation choices
Alternative:
4. a. Team name is already in use AND NOT Students List has students already in team
4. a. 1. TMSystem displays duplicate name error message
4. a. 2. AFTER 60 sec GOTO 2
4. b. Students List has students already in team AND NOT Team name is already in use
4. b. 1. TMSystem displays student already in team error message
4. b. 2. AFTER 60 sec GOTO 2
4. c. Team name is already in use AND Students List has students already in team
4. c. 1. TMSystem displays duplicate name, students in list error message
4. c. 2. AFTER 60 sec GOTO 2
Success Post-condition: Student is logged in AND TMSystem is ON AND Teams has team added AND TMSystem Display is student operation choices.

7.3.2.3 Use Case: Quit Team

Title: Quit team
Scope: Design
Level: user
Primary Actor: Student
Pre-condition: TMSystem is ON AND Student is logged in AND (Student status is member of team OR Student Status is pending)
Steps:
1. Student opens team in which she is member
2. TMSystem displays selected team information
3. Student choses quit team operation

4. TMSystem removes student from team
5. TMSystem sends notification removed member
6. TMSystem displays student operation choices
Success Post-condition: Student is logged in AND TMSystem is ON AND Student status is NOT member of team

7.3.2.4 Use Case: Join Team

Title: Join team
Scope: Design
Level: user
Primary Actor: Student
Pre-condition: TMSystem is ON AND Student is logged in AND Student status is NOT member of team
Steps:
1. Student chooses to visualize not complete students teams
2. TMSystem displays list of not complete teams
3. Student selects teams that she wants to join
4. TMSystem adds students to lists of candidate members
5. TMSystem displays student operation choices
Success Post-condition: TMSystem is ON AND Student is logged in AND Student status is pending

7.3.2.5 Use Case: Accept New Students

Title: Accept new students
Scope: Design
Level: user
Primary Actor: Liaison
Pre-condition: Liaison is logged in AND Student status is pending AND TMSystem is ON
Steps:
1. Liaison chooses to visualize list of students who asked to be members
2. TMSystem displays list of students
3. Liaison selects student to accept
4. TMSystem add student to team
5. TMSystem sends notification to added member
Success Post-condition: Liaison is logged in AND TMSystem is ON AND Student status is member of team

7.3.2.6 Use Case: Student Logout

Title: Student log out
Scope: Design
Level: user
Primary Actor: Student
Pre-condition: Student is logged in AND TMSystem is ON
Steps:
1. Student logs out
2. TMSystem displays log out acknowledgment
Success Post-condition: Student is not logged in AND TMSystem is ON

7.4 Global Combined Graph Generation

After importing all use case descriptions and the TMSystem use case model to UCEd, precede relations and resume operations can be captured by using the sequential relation capture tool. In the TMSystem use case model (refer to Figure 7-1), all use cases are main use cases. There are no extend, include and generalization relationships in the TMSystem use case model. Given the size of the generated global combined graph, we use a table as a clear way to describe the graph. Figure 7-2 presents the table. All precede relations are captured and resume operations are added in each use case description.

Main use case	Use case resumed from normal steps	Use case resumed from alternative steps
Instructor Login	Visualize students teams Setup Parameters Instructor Logout	N/A
Setup Parameters	Instructor Logout Visualize student teams	N/A
Modify Parameters	Instructor Logout Setup Parameters Visualize students teams	Instructor Logout
Visualize students teams	Instructor Logout Add member to a team Modify a parameter for a team Remove member from a team	N/A
Modify a parameter for a team	Visualize students teams Setup Parameters Instructor Logout	N/A
Remove member from a team	Setup Parameters Visualize students teams Instructor Logout	N/A
Add member to a team	Visualize students teams Instructor Logout Setup Parameters	N/A
Instructor Logout	Instructor Login	N/A
Student Login	Student Logout	N/A
Create team	Student Logout	N/A
Quit team	Join team Student Logout	N/A
Join team	Student Logout Quit team	N/A
Accept new students	N/A	N/A
Student Logout	Student Login	N/A

Figure 7-2: Global combined graph of Team Management System

7.5 Test Scenario Generation

In this section, a partial global combined graph is created to perform the process of test scenario generation. Figure 7-3 presents the partial combined global graph based on the TMSystem use case model (refer to Figure 7-1). Figure 7-5 shows the view of use case model-editing panel after the precede relations are added. Two use-case sequences:

1. Instructor login $\xrightarrow{precede}$ Setup Parameters $\xrightarrow{precede}$ Visualize student teams $\xrightarrow{precede}$ Add member to a team $\xrightarrow{precede}$ Instructor Logout, and
2. Join team $\xrightarrow{precede}$ Quit team $\xrightarrow{precede}$ Student Logout

can be obtained from the partial global combined graph. The first use-case sequence is initiated by the actor "Instructor", and the second use-case sequence is initiated by the actor "Student".

Figure 7-3: A partial global combined graph based on use case model in Figure 7-1

Figure 7-4 is the generated UCEd statechart for the first use-case sequence. The startchart is derived from the control flow-based state machines (abbr., CFSM) in order to convey the information clearly, in case sometimes there are too many nodes. However, path sequences are generated by traversing the "hidden" CFSMs. Test scenarios are then inferred from these path sequences and stored in the scenario model.

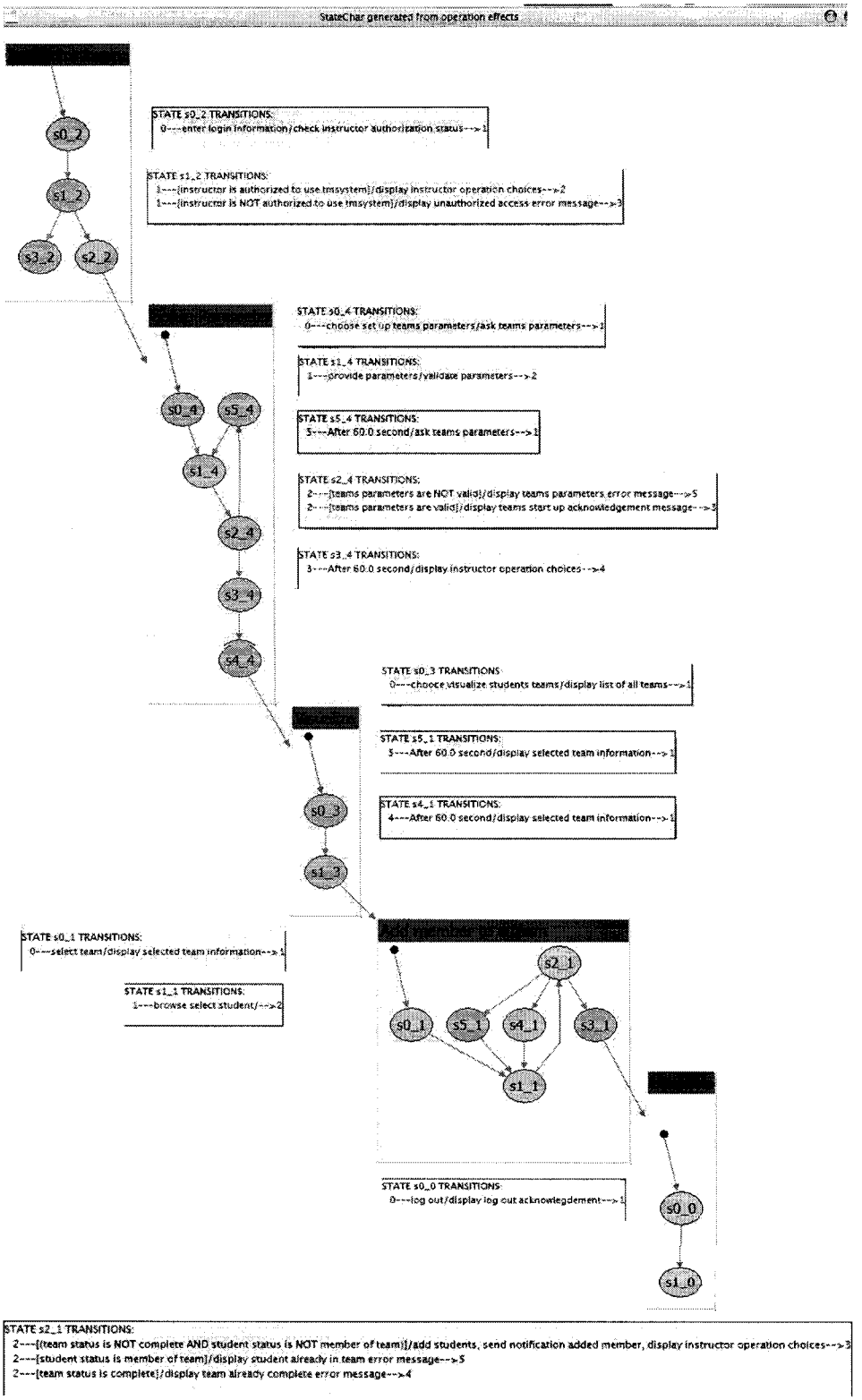


Figure 7-4: UCEd statechart generated for the first use-case sequence (see Appendix VI, transition information)

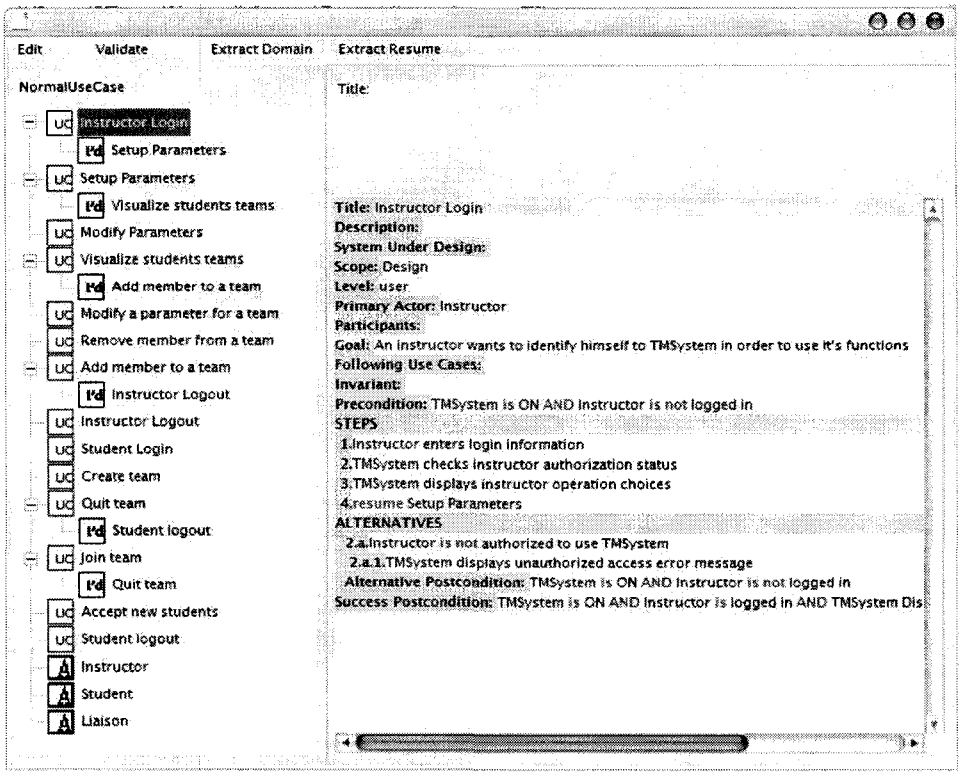


Figure 7-5: View of use case model-editing panel with four precede relations added in

According to the complete-path test coverage criterion, there will be *9* path sequences generated from the CFSM in Figure 7-4, which corresponds to the use-case sequence "Instructor login $\xrightarrow{precede}$ Setup Parameters $\xrightarrow{precede}$ Visualize student teams $\xrightarrow{precede}$ Add member to a team $\xrightarrow{precede}$ Instructor Logout". There will be *1* path sequence generated from the CFSM corresponding to the use-case sequence "Join team $\xrightarrow{precede}$ Quit team $\xrightarrow{precede}$ Student Logout" because there are no alternatives in the use case: "Join team", "Quit team" and "Student Logout". As there is only one path sequence generated for the second use-case sequence, test coverage is not required. However, the test coverage can be applied on generating path sequences for the first use-case sequence. The all-node coverage test scenarios are listed in Appendix IV and all-edge coverage test scenarios are listed in Appendix V.

7.6 Performance Analysis

We analyze the performance of the test-scenario generation tool of UCEd on a Pentium Centrino 1.6 PC with 512M memory. The performance analysis will be divided into two parts: path sequence generation and complete-path test coverage.

7.6.1 Performance on Path Sequence Generation

The core algorithm of path sequence generation is presented in section 6.3. As performance may vary on different hardware, we only consider a relative comparison. There are four parameters proposed, which will affect the time spent on generating path sequences.

Parameter 1: (Maximum) Depth of use-case sequence

The depth of a use-case sequence is the (maximum) number of precede relationships in a use-case sequence beginning with a certain use case. For instance, the depth of a use-case sequence: "Instructor Login $\xrightarrow{precede}$ Setup Parameters $\xrightarrow{precede}$ Visualize students teams $\xrightarrow{precede}$ Modify a parameter for a team $\xrightarrow{precede}$ Instructor Logout" is 4. All use cases in a use-case sequence are main use cases. As more use cases are concatenated to a use-case sequence where it is applicable, the CFSMs generated from the new use cases will be combined into the original CFSM. As previously discussed, the combination of two CFSMs is that a transition is added (i.e., precede relation) beginning from the leaf node of a CFSM (i.e., generated from the original use case) to the initial node of another CFSM (i.e, generated from the succedent use case of the original use case). In addition, the succedent use case that is resumed from the original use case may have its own succedent use cases. As the number of the CFSMs increase, the traversal will be able to obtain more path sequences. A new CFSM will bring more nodes and edges thus new combination of these edges will create new path sequences. Therefore, the depth of use-case sequence affects the length of path sequence. It should be considered as a parameter.

Parameter 2: (Maximum) Breadth of a use case

The breadth of a use case is the (maximum) number of succedent use cases of a main use case. For instance, if the depth of a use case is set to *1*, the breath of the use case: "Instructor Login" is *3* and the breath of the use case: "Setup Parameters" is *2*. If the depth of a use case is set to more than *1*,

the maximum breadth can be calculated by creating a tree. The number of the tree-leaf nodes is the maximum breadth of the main use case (i.e., the root of the tree). Figure 7-6 presents a tree from the main use case: "Instructor Login". Its maximum breadth is the number of the leaf nodes, which is *10*. The breadth is the ability to contain succedent use cases of a main use case. Similar to the depth of a use-case sequence, as the breadth of a use case increase, more CFSMs will be combined. Therefore, breadth of a use case should be considered as a parameter.

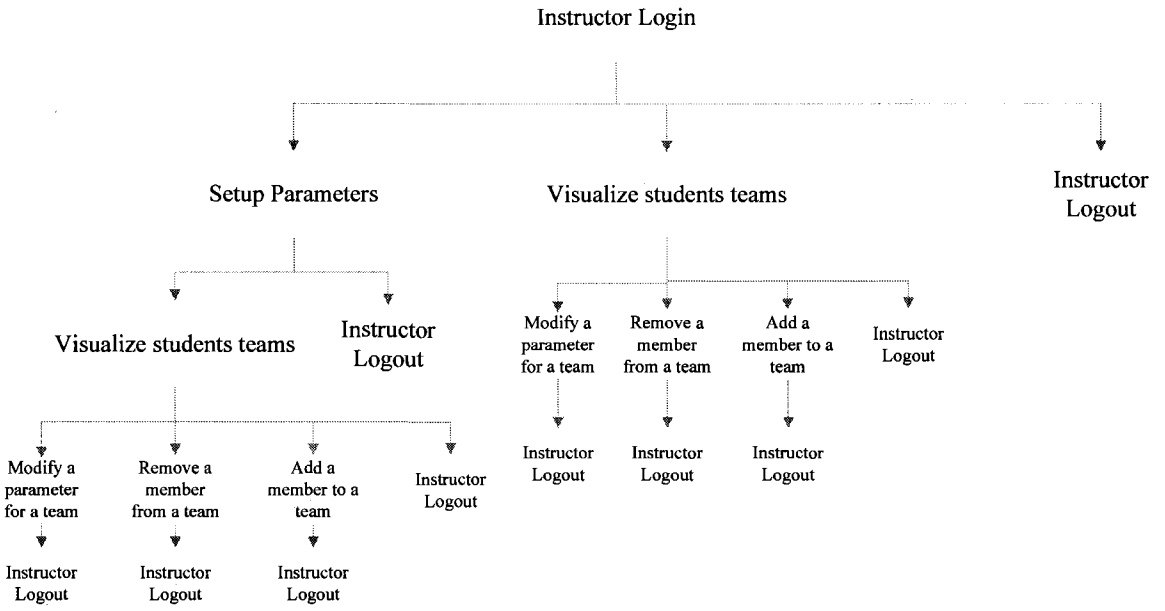


Figure 7-6: A tree view of succedent use cases of the main use case: "Instructor Login"

Parameter 3: The number of edges

With the above two parameters in mind, it is easy to take into account the number of edges as a parameter. A single path sequence consists of many edges. Each edge represents one of the following, a trigger, a system reaction, a delay or a guard condition in the generated test scenario. As discussed in previous chapters, as more CFSMs are combined, longer path sequences can be generated. In the depth-first traversal algorithm, if the path sequence contains more edges, it will take a longer time to be built. Therefore, the number of edges should be considered as a parameter.

Parameter 4: The number of path sequences

The number of the generated path sequences is definitely a parameter for performance analysis. As the number of path sequences increases, it will take longer to list them.

We set up the following experiments to see the time results by applying different parameters. Parameter *1* and parameter *2* can be set up at the beginning of the experiment while parameter *3* and parameter *4* will be obtained at the end of the experiment. We use different curve lines to denote different depth. Figure 7-7 presents a curve-line graph. Four values of the depth are denoted on the curve-line graph from *1* to *4*. The X-axis represents the breadth and the Y-axis represents the time elapsed. The time is calculated in milliseconds. Table 7-1 is a table recording the data of this experiment. The number in the bracket is the number of path sequences and the maximum number of edges from a single path sequence. As the depth is deeper and breadth is broader, the time of the generated path sequences grows. The number of edges and the number of path sequences have an impact on the time elapsed. For instance, the time elapsed for the case of depth is *3* and breadth is *6* is *291*, yet the time elapsed for the case of depth is *4* and breadth is *6* is *260*. It is because the edge number of the first is *25* while the edge number of the second is *14*.

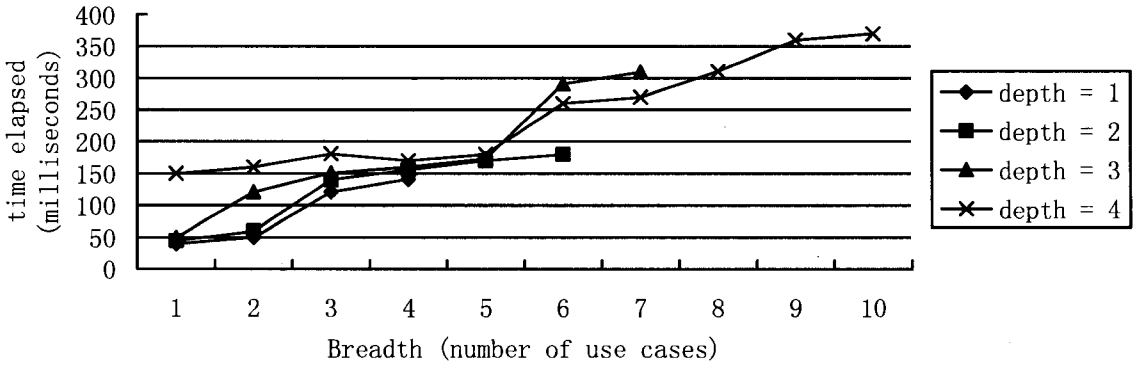


Figure 7-7: Curve-line graph for analyzing path sequence generation

Depth/Breadth	1	2	3	4	5	6	7	8	9	10
1	40 (3/19)	50 (4/19)	121 (4/19)	141 (5/19)						
2	45 (3/21)	60 (4/21)	140 (5/21)	156 (7/21)	170 (10/25)	180 (12/25)				
3	50 (3/23)	121 (4/23)	151 (5/23)	160 (7/23)	173 (13/40)	291 (25/40)	310 (28/40)			
4	150 (5/42)	160 (6/42)	181 (8/42)	170 (10/42)	180 (11/42)	260 (14/42)	270 (23/42)	311 (25/42)	360 (26/42)	370 (28/42)

Table 7-1: Four parameters that influence the path sequence generation performance

7.6.2 Performance on Branch Statement

Recall that a branch statement is a "GOTO" statement in use case descriptions. It appears at the last step of certain alternatives so that a recovery scenario is recovered to achieve the main success scenario goal at last. The branch statement is regarded as a loop in the generated path sequences. According to complete-path test coverage criterion, a user can select one or more times to execute such loops. The times to execute the branch statement are restricted by a repetition limit. In this section, a performance analysis is conducted under different number of loops.

An experiment is set up and repeated ten times, with a loop appearing from one to ten respectively. This means that for each time, the depth and breath of a main use case do not change. The number of path sequences does not change, yet the edge number will be augmented. The repetition limit is from *1* to *10*. For each repetition limit, *5* experiments are carried out and they are indicated in experiment *1* to experiment *5*. Figure 7-8 presents a curve-line graph denoting the result of our experiment. Table 7-2 is the result data of the experiment. We observe that as the repetition parameter grows, the interval grows. In this experiment, only the edge number is augmented. Therefore, it also proves that the edge number is a parameter that affects the performance of path sequence generation.

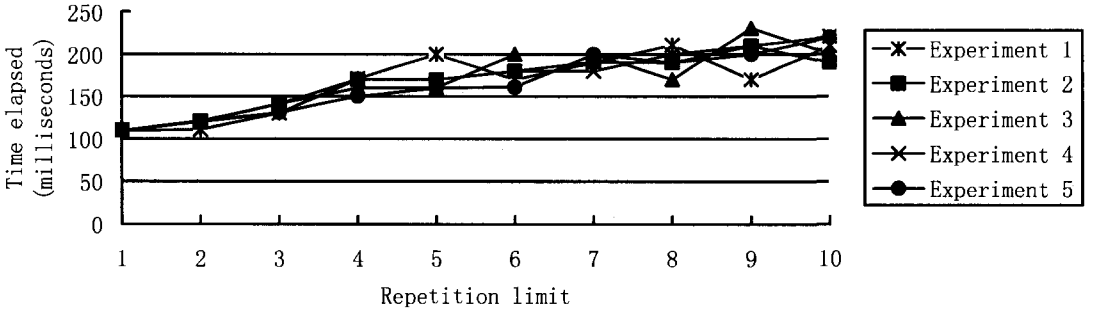


Figure 7-8: Curve-line graph for analyzing different repetition limit

Experiment/Repetition	1	2	3	4	5	6	7	8	9	10
1	110	111	130	171	200	170	190	211	170	210
2	110	121	141	170	170	180	191	190	210	190
3	110	120	141	160	160	200	200	180	231	201
4	110	120	130	170	170	180	180	200	210	221
5	110	121	131	150	160	161	200	190	200	220
Average	110	118.6	134.6	164.2	172	178.2	192.2	194.2	204.2	208.4

Table 7-2: Repetition limit experiment result data

7.7 Chapter Summary and Highlights

In this chapter, a Team Management System is built to perform a case study. Users are not encouraged to add all precede relations by using the sequential relation capture tool. Instead, they are encouraged to add the precede relations about the functionalities that they are willing to test. Through this analysis, four parameters (i.e., depth of use-case sequence, breadth of a use case, the number of edges and the number of generated path sequences) that will have impact on the performance of the test scenario generation tool are proposed. The four parameters are closely related. The deeper the depth of use-case sequence, the more use cases of the main use case, which begins the use-case sequence, will have. In addition, the edges and path sequences will be more. Therefore, it will take more time to complete each task of path sequence generation.

Another experiment is set up to analyze the branch statement. It is also a performance analysis analyzing the complete-path test coverage criterion. The complete-path test coverage criterion has the strongest testing criterion; however, it is not practical for a tester. There are many paths between the entry and exit nodes. Adding even a few simple decision statements increases the number of paths. In addition, every loop multiplies the number of paths based on the number of possible iterations of the loop since each iteration constitutes a different path. The more the loop appears, the longer it will take.

Chapter 8 – CONCLUSIONS

The sequential relation capture tool and test scenario generation tool are implemented and integrated into UCEd based on the approach proposed in this thesis. Our approach is now able to perform a new process from requirement specification to system enhancement and then conduct test case creation. This chapter reviews the approaches and emphasizes the contributions. It also discusses limitations and future research arising from several issues encountered during the research.

8.1 Contributions

In a software development life cycle (abbr., SDLC), test plans and test cases are usually written when detail designs are finished. The test-case writing is arbitrary and manually. For the purpose of our research, we propose that the test-case writing could be shifted to the requirement phase. Therefore, in addition to testers, both business analysts and software developers can create test cases. The objective of our research is to promote more accurate detail designs. In addition, the whole SDLC will benefit from that, not only in recovered time, but also in the quality of the software product itself.

To summarize, the approach presented in the thesis can be divided into five steps, noted as below:

1. The use case descriptions are well-written in Cockburn's use-case format in UCEd and validated fine.
2. The use case descriptions are enriched by using sequential relation capture tool to add resume operations and precede relations.
3. The control flow-based state machines (abbr., CFSM) of different use cases are combined by integrating new transitions corresponding to these precede relations.
4. The path sequences are generated by traversing these combined CFSMs based on certain test coverage.
5. The generated path sequences are mapped to test scenarios by recognizing triggers, system reactions, assertions, guards and delays.
6. The test scenarios as well as the use case descriptions can be mapped to concrete test cases by recognizing test conditions, test data assumption, input, guard, expected results and test oracles.

Based on the approach presentation, the thesis makes three contributions. These contributions are implemented by supporting test case generation for UCEd.

- ***First Contribution: Extraction of implicit use case sequential relations***

  There are various formats for writing use cases at different levels of abstraction. Wirfs-Brock, Cockburn and Essential use cases are formats for use case writing at a detailed black-box level. Cockburn suggests that use cases be written using a restricted form of language. In addition, several authors recommended various use cases guidelines for writing restricted forms of use case descriptions. Our approach is based on Cockburn's use cases. Users can determine the use case sequential relations that are not bound by UML use case relationships with their own thoughts. The process is, however, manual and very time-consuming. Hence, we propose an automated approach to capture the sequential relations. The capture begins with comparing pre-conditions and post-conditions based on predicates. Then, a sequential relation capture tool was implemented to provide a flexible solution for the users to update sequential relations in real use case models. By using the tool, sequential relations can be captured accurately and automatically.

- ***Second Contribution: Test scenarios are automatically generated from control flow-based state machines with some test coverage.***

  CFSMs can be generated from use cases. A CFSM formalizes and integrates a set of related use cases. Different CFSMs can be combined thus are enhanced to a system level by using the inferred sequential relations. Test scenarios are generated directly and fully automatically from the system-level CFSMs by applying a depth-first traversal algorithm according to some test coverage criteria. These generated test scenarios can be further applied to create concrete test cases. This approach was implemented in a test scenario generation tool and it was integrated into UCEd.

- ***Third Contribution: Automated scenario generation and simulation in the UCEd approach***

  By integrating the test scenario generation tool, UCED scenarios are automatically generated according to certain test coverage criteria. They are then stored in the scenario model. Users can use the scenario-editing tool to edit the test scenarios. Originally, scenarios were used in the

UCEd approach to validate use case requirements using simulation. Users are required to input scenario elements and create scenarios based on their own thoughts. Thus, writing scenarios for testing is in an arbitrary manner and the users can never determine when the coverage is satisfied. We generate UCEd scenarios automatically from our test scenarios. This allows a better coverage of use cases during the validation. The new automated test scenarios generation and simulation save a lot of time and is more strict.

8.2 Limitations and Future Work

With the contribution of the thesis proposed, the limitations should be proposed as well as the future research. The approach involves the following three limitations and we propose corresponding future research.

8.2.1 Test Data Combination Automation

- ***Limitation***

  The integration of state machines exhibits a limitation of not being able to involve the test data into the test scenarios. In this case, test data has to be added in to the test scenarios manually to generate concrete test cases. Normally, the test data consists of numeric or alphabetic forms. Sample test data should conform to a test boundary rule. Domain models and databases can be applied to store a great deal of test data. These test data will be combined to guard conditions, delays, triggers and system reactions correspondingly.

- ***Future work***

  To automate the process, a parser for test data generation is required. This parser should be able to distinguish different kinds of test data and recognize that which step in the test scenarios it belongs. As proposed in section 6.5, test setup contains test data assumptions. These test data assumptions can be obtained from guard condition in test scenarios. The parser should be able to categorize valid test data and corresponding invalid test data from a data repository (e.g., domain model or database) according to guard conditions. Thus, once a test scenario is automatically generated, some sample test data will be combined automatically too. When the process can be automated in the future, the whole process from requirement to test case generation will be automated. Moreover, besides condition coverage which are based on control

flows, data flow test coverage can be applied to guide the test data generation, such as all-def, all p-uses, all-uses, all def-use paths etc.

8.2.2 Post-condition Tracing Automation from Pre-condition

- ***Limitation***

  Another limitation is that, as our approach is based on use case descriptions and use case diagrams, it can only capture the implicit sequential relations from the pre-condition and post-condition information that users have provided. The tool is not able to capture the implicit sequential relations that the users missed.

- ***Future work***

  In the future, a mechanism to trace the use case description will be provided. For each step in a use case description, a post-condition will automatically be generated from a pre-condition. For instance, the step *1* in the use case: "Customer Login" is "OPSystem asks customer name", its pre-condition "OPSystem is ON AND Customer is not logged in" is the pre-condition of this use case. The tracing will automatically trace to monitor if the pre-condition is changed after executing this step. Thus, the tracing will provide a post-condition after all the steps have been executed. Such a mechanism will convey a strict way of writing pre-conditions and post-conditions. Moreover, as reviewed in section 3.3, inadmissible scenarios can be captured when the users write the steps in an unpractical order.

8.2.3 UCEd Generalization Relationship Enhancement

- ***Limitation***

  The current UCEd release does not support generalization relationships. However, this relationship is effective to describe an "is-a-kind-of" relation. In the use case model of the OPSystem, users have two choices (i.e., credit card payment and cheque payment) to make the payment. When generating test cases, different test scenarios will be generated for each of them. Although our current approach can realize the generation by replacing the use case: "Credit Card Payment" to the use case: "Cheque Payment", the process is time-consuming.

- ***Future work***

  In the future, the generalization relationships will be implemented in the UCEd. When

generating test scenarios, the UCEd will be able to recognize abstract points and specialization conditions. Thus, the CFSMs will be generated considering the generation relationships according to our approach proposed in section 5.3.1 and section 6.2.1. When the traversal from a parent use case reaches the abstract point, the corresponding specialization conditions will be added, thus generated test scenarios will contain the generalization relationship information.

BIBIOLOGRAPHY

[Arm01] F.Armour and G.Miller, "Advanced use case modeling software systems", Addison Wesley, 2001

[Bec02] P. Becker. "Eliminating functional defects through model-based testing", 2002. available online (Oct 2006) at: http://www.stickyminds.com

[Bei95] B. Beizer. "Black-box Testing: Techniques for functional testing of software and systems", John Wiley & Sons, Inc, New York, 1995

[Bin99] R. V. Binder, "Testing Object-Oriented Systems - Models, Patterns, and Tools", Addison-Wesley, 1999.

[Boe81] B. Boehm, "Software Engineering Economics", Englewood Cliffs, N.J., Prentice Hall, 1981.

[Bri02] L. C. Briand, Y. Labiche. "A UML-based approach to system testing", Software and System Modeling, 1(1):10-42, 2002.

[Bri05] L. C. Briand, Y. Labiche, Y. Lin, "Improving Statechart Testing Criteria Using Data Flow Information", Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on 08-11 Nov. 2005 Page(s):95 - 104

[Bru94] B. A. Cota, D. G. Fritz, R. G. Sargent, "Control flow graphs as a representation language", Proceedings of the 26th conference on Winter simulation, December 1994

[Bur02] I. Burnstein, "Practical Software Testing: a process-oriented approach", Springer, 2002

[Cav04] A. Cavarra, C. Crichton, J. Davies. "A method for the automatic generation of test suites from object models", Information & Software Technology, 46(5):309-314, 2004.

[Coc01] A. Cockburn. "Writing Effective Use Cases", Addison Wesley, 2001.

[Coc95] A. Cockburn, "Structuring Use Cases with goals", Humans and Technology, HaT. Technical Report. 1995, available online (Oct 2006) at: http://alistair.cockburn.us/crystal/articles/sucwg/structuringucswithgoals.htm

[Con99] L. L. Constantine and L. A. D. Lockwood, "Software for Use: A Practical Guide to the Model and Methods of Usage Centered Design", Addison-Wesley, 1999

[Cri01] C. Crichton, A. Cavarra, Jim Davies, "Using UML for Automatic Test Generation", Paper submitted to ASE (Automated Software Engineering) conference, 2001.

[Ext97] J. D. Wells, "Extreme Programming: A Gentle Introduction", available online (May 2006) at: http://www.extremeprogramming.org

[Frö00] P. Fröhlich and J. Link. "Automated test case generation from dynamic models", In E. Bertino, editor, ECOOP, volume 1850 of Lecture Notes in Computer Science, pages 472-492. Springer, 2000.

[Fow98] M. Fowler, "Use and abuse cases", Distributed computing, April 1998.

[Har87] D. Harel. "Statecharts: A visual formalism for complex systems", Science of Computer Programming, 8(3):231–274, 1987.

[Hou98] D. Houston, J. B. Keats, "Cost of Software Quality: A Means of Promoting Software Process Improvement", Quality Engineering, Vol. 10, No. 3, MARCH 1998, pp. 563-573

[Hüb03] M. Hübner, I. Philippow, M. Riebisch, "Statistical Usage Testing Based on UML", In: Nagib Callos et al. (Eds.): Proc. 7th World Multiconferences on Systemics, Cybernetics and Informatics, July 27-30, 2003, pp. 290-295

[Iso05] S. Isoda, "Improving UML's Definition of the Use-Case Class", Systems and Computers in Japan, Volume 36, Issue 6 (p 14-25), 2005

[Jac92] I. Jacobson, M. Christerson, P. Johnsson, G.Overgaard, "Object-Oriented Software Engineering: A Use Case Driven Approach", Addison-Wesley, Wokingham, 1992

[Jac99] I. Jacobson, G. Booch, J. Rumbaugh, "The Unified Software Development Process", Addison-Wesley, 1999

[Jac05] I. Jacobson, P. W. NG, "Aspect-Oriented Software Development with Use Cases", Addison-Wesley, 2005

[Jor95] P. C. Jorgrnsen, "Software Testing: A Craftsman's Approach", CRC PressInc 1995

[Kan03] S. Kansomkeat, W. Rivepiboon, "Automated-generating test case using UML statechart diagrams", Proceedings of the 2003 annual research conference of the SAICSIT '03 on Enablement through technology, September 2003

[Kan05a] J. Kanyaru, K. Phalp, "Supporting the Consideration of Dependencies in Use Case Specifications", 11th International Workshop on Requirements Engineering: Foundation For Software Quality - REFSQ´05, Porto, Portugal, 13-14 June 2005

[Kan05b] J. Kanyaru, K. Phalp, "Requirements validation with enactable models of state-based use cases", Empirical Assessment in Software Engineering, EASE 2005, Keele University, 11-13 April 2005

[Kor90] B. Korel, "Automated Software Test Data Generation", IEEE Transactions on Software Engineering, Vol. 16, No. 8, pages 870-879, August 1990.

[Kra90] H.Krasner, "Self-Assessment Experiences at Lockheed", Proceedings of the SEI/AIAA Software Process Improvement Workshop, Chantilly, Va., Nov. 8, 1990.

[Lar05] L. Craig 2005. "Applying UML and Patterns: An Introduction to Object-Oriented Analysis

and Design", 3rd edition, Prentice Hall, Upper Saddle River, NJ

[Mil63] J. C. Miller, C. J. Maloney, "Systematic mistake analysis of digital computer programs", Communications ACM, 1963, Pages: 58 – 63

[Neb03] C. Nebut, F. Fleurey, Y. L. Traon, J-M. Jézéquel, "Requirements by Contracts allow Automated System Testing", In proceedings of 14th international symposium on software reliability engineering, 2003(ISSRE '03)

[Ngu03] H. Q. Nguyen, B. Johnson, M. Hackett, "Testing Application on the Web: Test Planning for Internet-based systems", John Wiley&Sons, 2003

[OMG99] OMG, Unified modeling language specification, 1999, available online (Oct 2006) at: http://www.omg.org/technology/uml/index.htm

[OMG03] OMG UML Revision Task Force. OMG Unified Modeling Language Specification v. 1.5. Document formal/03-03-01. Object Management Group; 2003.

[OMG03b] Object Management Group. Unified Modeling Language Superstructure Specification, August 2003, ptc/03-08-02.

[OMG05] Object Management Group, Unified Modeling Language Superstructure Specification, v2.0, Document – formal/05-07-04

[Pha03] K. Phalp, K. Cox, "Using Enactable Models to Enhance Use Case Descriptions", ProSim'03, International Workshop on Software Process Simulation Modelling (in conjunction with ICSE 2003), Portland, USA, May 3-4 2003.

[Pic00] S. Pickin, J.-M. Jezequel, "Using UML sequence diagrams as the basis for a formal test description language". In Proc. of the 4th International Conference on Integrated Formal Methods (IFM), April 2000.

[Rie05] M. Riebisch, M. Huebner, "Traceability-Driven Model Refinement for Test Case Generation". In Proc. of the 12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS), April 2005.

[Rie03] M. Riebisch, I. Philippow, M. Götze: "UML-Based Statistical Test Case Generation". in: Mehmet Aksit, Mira Mezini, Rainer Unland (Eds.): Objects, Components, Architectures, Services and Applications for a Networked World. LNCS 2591. Springer 2003, pp. 394-411.

[Rys99] J. Ryser, M. Glinz. "SCENT: A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts". In Proc. 12th International Conference on Software and Systems Engineering and their Applications, Dec 1999.

[Rys00] J. Ryser, M. Glinz, "Using Dependency Charts to Improve Scenario-Based Testing", Proc. of TCS2000 Washington D.C., June 2000.

[Sch98] G. Schneider, J. P. Winters. "Applying Use Cases a practical guide", Addison-Wesley, 1998.

[Som03] S. Somé, "An approach for the synthesis of state transition graphs from use cases". In Proceedings of the International Conference on Software Engineering Research and Practice (SERP'03), volume I, pages 456-462, june 2003.

[Som04b]S.Somé. "Supporting use cases based requirements simulation". In Proceedings of the International Conference on Software Engineering Research and Practice (SERP'04), volume I, pages 381-386, June 2004.

[Som05a]S. Somé. "Enhancement of a Use Cases based Requirements Engineering approach with Scenarios". In Proceedings of the 12th Asia Paci_c Software Engineering Conference (APSEC 2005), Dec 2005.

[Som05b]S.Somé, "Synthesis of Statecharts from Use Cases", 2005

[Som05d]S.Somé, UCEd User Guide, available online (Oct 2006) at: http://www.site.uottawa.ca/~ssome/publis/userGuide1_6/index.html

[Som06] S.Somé. "Supporting use case based requirements engineering". Information and Software Technology 48(2006) 43-58

[Sou99] D. Souza, A.Wills. "Objects, Component, and Frameworks with UML, The Catalysis Approach", Chapter Interaction Models: Use cases, Actions, and collaborations. Addison-Wesley, 1999

[Wan04] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, G. Zheng. "Generating test cases from UML activity diagram based on gray-box method". In APSEC, pages 284-291. IEEE Computer Society, 2004.

[Wif01] R.Wirfs-Brock, J.Schwartz, "The Art of Writing Use Cases, Wirfs-Brock Associates, Inc." 2001, http://www.cs.joensuu.fi/pages/ageenko/teaching/OOD/Use_Cases.pdf access online on April 2006.

[Wit01a] J. Wittevrongel, F. Maurer: "Using UML to Partially Automate Generation of Scenario-Based Test Drivers". OOIS 2001: 303-306

[Wit01b] J. Wittevrongel, F. Maurer, "SCENTOR: Scenario-Based Testing of E-Business Applications," Proc. 10th Int. Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2001, p. 41.

Appendix I: Order Process System Use Case Templates without resume operation captured

Title: Turn ON System
Primary Actor: Customer
Secondary Actor/Participants: Order Process System
Goal: The customer wants to turn on the system.
Pre-condition: OPSystem is OFF AND Customer is not logged in AND Sales Clerk is not logged in
Steps:
1. Customer/Sales Clerk turns on the OPSystem
Alternatives:
None
Success Post-condition: OPSystem is ON AND Customer is not logged in AND Sales Clerk is not logged in

Figure I-1: Use Case – Turn ON System

Title: Customer Login
Primary Actor: Customer
Secondary Actor/Participants: Order Process System
Goal: A customer wants to identify him/her in order to use the order process system to place the order.
Pre-condition: OPSystem is ON AND Customer is not logged in
Steps:
1. OPSystem asks customer name
2. Customer types the name and confirm
3. OPSystem asks customer password
4. Customer types the password and confirm
5. OPSystem displays a welcome message
6. OPSystem displays a home page of the system
Alternatives:
2a. Customer name is not exist
  2a1. OPSystem shows a customer name nonsexist error message
  2a2. Goto Step 1
2b. Customer name is illegal format
  2b1. OPSystem shows a customer name illegal format error message
  2b2. Goto Step 1
4a. Customer password is incorrect
  4a1. OPSystem shows a customer password incorrect error message
  4a2. Goto Step 3
Success post-condition: Customer is logged in AND OPSystem is ON

Figure I-2: Use Case – Customer Login

Title: Sales Clerk Login
Primary Actor: Sales Clerk
Secondary Actor/Participants: Order Process System
Goal: A Sales Clerk wants to identify him/her in order to use the order process system to process the order.
Pre-condition: OPSystem is ON AND Sales Clerk is not logged in
Steps:
1. OPSystem asks for sales clerk name
2. Sales clerk types the sales clerk name and confirm
3. OPSystem asks for sales clerk password
4. Sales clerk types the password and confirm
5. OPSystem displays a welcome message
6. OPSystem displays the homepage
Alternatives:
2a. Sales clerk name is not exist
  2a1. OPSystem shows a sales clerk name nonexist error message
  2a2: Goto Step 1
2b: Sales clerk name is illegal format
  2b1. OPSystem shows a sales clerk name illegal format error message
  2b2. Goto Step 1
4a Sales clerk password is incorrect
  4a1. OPSystem shows a sales clerk password incorrect error message
  4a2. Goto Step 3
Success post-condition: Sales Clerk is logged in AND OPSystem is ON

Figure I-3: Use Case – Sales Clerk Login

Title: Process Order
Primary Actor: Sales Clerk
Secondary Actor/Participants: Order Process System
Goal: A Sales Clerk wants to process the order in order to send out the item which the customer requested in the order.
Pre-condition: Order is placed AND OPSystem is ON AND Sales Clerk is logged in
Steps:
1. Sales clerk chooses the order
2. OPSystem lists the orders
3. Sales clerk requests warehouse to send the items.
4. OPSystem displays confirm message
Alternatives:
None
Success post-condition: Order is processed AND Sales Clerk is logged in AND OPSystem is ON

Figure I-4: Use Case – Process Order

Title: Place Order
Primary Actor: Customer
Secondary Actor/Participants: Order Process System
Goal: Customer can place orders and pay the order by credit card payment
Pre-condition: Order System is ON AND (Customer is logged in OR Sales Clerk is logged in)
Steps:
1. Customer selects items
   Extension point ➔ Item selected
2. OPSystem identifies items
3. Customer types item number
   Extension point ➔ Item number typed
4. OPSystem validates item number.
5. OPSystem displays successful message
5: Include Update Account
6. Include Credit Card Payment.
Alternatives:
4a. Item number is invalid
 4a1. OPSystem shows item number invalid error message
 4a2 Goto Step 3
Success Post-condition: Order is placed AND OPSystem is ON AND (Customer is logged in OR Sales Clerk is logged in)

Figure I-5: Use Case – Place Order

Title: Cancel Order
Primary Actor: Customer
Secondary Actor/Participants: Order Process System
Goal: A customer wants to cancel his order because it is on hold.
Pre-condition: Customer is logged in AND OPSystem is ON AND Order is placed OR Order is on hold.
Steps:
1. Customer selects the on hold orders.
2. Customer chooses to cancel the orders.
3. Order is cancelled
4. OPSystem displays success message.
Alternatives:
None
Success Post-condition: Order is cancelled AND Customer is logged in AND OPSystem is ON

Figure I-6: Use Case – Cancel Order

Title: Update Order
Primary Actor: Customer
Secondary Actor/Participants: Order Process System
Goal: A customer wants to update his orders because it is on hold.
Pre-condition: Customer is logged in AND OPSystem is ON AND Order is placed
Steps:
1. Customer requests the OPSystem to list the on-hold orders.
2. OPSystem lists placed orders.
3. Customer changes the item number
4. Customer submits the orders.
5. OPSystem display success message
6. Include update account.
Alternatives:
3a If item number is invalid
  3a1. Order is on hold
  Alternative Post-conditions: Customer is logged in AND OPSystem is ON AND Order is on hold.
Success Post-condition: Customer is logged in AND OPSystem is ON AND Order is updated AND Order is

Figure I-7: Use Case – Update Order

Title: Update Account
Primary Actor: Customer
Secondary Actor/Participants: Order Process System
Goal: A customer wants to update his/her account.
Pre-condition: Customer is logged in AND OPSystem is ON.
Steps:
1. Customer account information is listed.
2. Customer changes the customer account information
3. OPSystem verifies the customer account information
4. OPSystem displays success message
Alternatives:
3a If customer account information has typo
 3a1. OPSystem shows account information error message
 3a2. Goto Step 2.
Success Post-condition: Customer information is updated AND Customer is logged in AND OPSystem is ON

Figure I-8: Use Case – Update Account

Title: Credit Card Payment
Primary Actor: Customer
Secondary Actor/Participants: Order Process System
Goal: Customer wants to pay the payments by credit card
Pre-condition: OPSystem is ON AND Customer is logged in AND Order is placed
Steps:
1. Customer selects Credit Card Payment
2. Customer types credit card number
3. Customer types expiration date
4. OPSystem validates credit card information
Alternatives:
4a Credit card information is invalid AND customer submit attempts < 4
 4a1 Goto Step 2
4b Credit card information is invalid AND customer submit attempts = 4
 4b1 OPSystem displays error message
 Alternative Post-condition: OPSystem is ON AND Customer is logger in AND Order is placed
Success Post-condition: OPSystem is ON AND Customer is logged in.

Figure I-9: Use Case – Credit Card Payment

Title: Request Catalog
Parts:
p1: At Extension Point item selected
 1p1 OPSystem mails requested catalog to the mailing address.
p2: At Extension Point item number typed
 1p2 OPSystem mails requested catalog to the mailing address

Figure I-10: Use Case – Request Catalog

Title: *Arrange Payment*
Primary Actor: Customer
Secondary Actor/Participants: Order Process System
Goal: Customer wants to select a Payment Method
Pre-condition: OPSystem is ON AND Customer is logged in AND Order is placed
Steps:
1. OPSystem displays Payment method selection
2. Customer selects Payment Method
3. OPSystem displays order number and record
4. OPSystem displays payment complete
Alternatives:
1a. After 60 seconds
 1a1. OPSystem alarm warning
 1a2. GOTO Step 1
Success Post-condition: Order payment is complete AND OPSystem is ON AND Customer is logged in.

Figure I-11: Use Case – Arrange Payment

Title: Cheque Payment
Primary Actor: Customer
Secondary Actor/Participants: Order Process System
Goal: Customer wants to pay the payments by cheque
Pre-condition: OPSystem is ON AND Customer is logged in AND Order is placed
Steps:
1. Customer selects Cheque payment
2. OPSystem puts order on hold
Success Post-condition: Order is on hold AND OPSystem is ON AND Customer is logged in.

Figure I-12: Use Case – Cheque Payment

Appendix II: Path Sequences Extracted from Partial Order Process System (Figure 6-3)

	Path Sequences – Complete-path coverage, repetition limit is 1
1	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-29-30
2	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-29-30
3	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35
4	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-32-33-34-35
5	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-24-25-33-34-35
6	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-29-30
7	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-29-30
8	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35
9	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-32-33-34-35
10	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-24-25-33-34-35
11	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-29-30
12	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-29-30
13	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-32-33-34-35
14	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35
15	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-24-25-33-34-35
16	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-29-30
17	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-29-30

18	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35
19	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-32-33-34-35
20	0-1-2-3-4-5-6-7-8-9-10-5-6-7-8-11-12-13-14-15-18-19-20-21-22-24-25-33-34-35
21	0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-29-30
22	0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-29-30
23	0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-32-33-34-35
24	0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35
25	0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-24-25-33-34-35
26	0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-29-30
27	0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-29-30
28	0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35
29	0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-32-33-34-35
30	0-1-2-3-4-5-6-7-8-11-12-13-14-15-18-19-20-21-22-24-25-33-34-35
31	0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-29-30
32	0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-29-30
33	0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-32-33-34-35
34	0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35
35	0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-24-25-33-34-35
36	0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-29-30
37	0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-29-30
38	0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35
39	0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-32-33-34-35
40	0-1-2-3-4-5-6-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-24-25-33-34-35
41	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-29-30

42	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-29-30
43	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-32-33-34-35
44	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35
45	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-24-25-33-34-35
46	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-29-30
47	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-29-30
48	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35
49	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-32-33-34-35
50	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-18-19-20-21-22-24-25-33-34-35
51	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-29-30
52	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-29-30
53	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-32-33-34-35
54	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35
55	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-24-25-33-34-35
56	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-29-30
57	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-29-30
58	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35
59	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-32-33-34-35
60	0-1-3-4-5-7-8-9-10-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-24-25-33-34-35
61	0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-29-30
62	0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-29-30
63	0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27-28-32-33-34-35
64	0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35

65	0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-36-37-21-22-24-25-33-34-35
66	0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-29-30
67	0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-29-30
68	0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35
69	0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-32-33-34-35
70	0-1-3-4-5-7-8-11-12-13-14-15-18-19-20-21-22-24-25-33-34-35
71	0-1-3-4-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27 -28-29-30
72	0-1-3-4-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-29-30
73	0-1-3-4-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-31-26-27 -28-32-33-34-35
74	0-1-3-4-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-23-26-27-28-32-33-34-35
75	0-1-3-4-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-36-37-21-22-24-25-33-34-35
76	0-1-3-4-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-29-30
77	0-1-3-4-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28-29-30
78	0-1-3-4-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-32-33-34-35
79	0-1-3-4-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-23-26-27-28-31-26-27-28 -32-33-34-35
80	0-1-3-4-5-7-8-11-12-13-14-15-16-17-13-14-15-18-19-20-21-22-24-25-33-34-35

Appendix III: Automatically Generated Test Scenarios (Order Process System)

Use case: "Place Order" with three succedent use cases: "Update Order", "Cancel Order", "Process Order" (All-Node Test Coverage)

Assertion: (OPSystem is ON AND (Customer is logged in OR Sales Clerk is logged in))
Trigger: customer select items
Guard: Item catalog is available
System Reaction: OPSystem mail requested catalog
System Reaction: OPSystem identify items
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is not invalid
System Reaction: OPSystem display successful message
Trigger: customer account information is listed
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has typo
System Reaction: OPSystem shows account information error message
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has not typo
System Reaction: OPSystem display success message
Trigger: customer type credit card number
Trigger: customer type expiration date
System Reaction: OPSystem validate credit card information
Guard: (Credit card information is invalid AND customer submit attempts < 4)
Trigger: customer type credit card number
Trigger: customer type expiration date
System Reaction: OPSystem validate credit card information
Guard: Credit card information is valid
System Reaction: OPSystem display payment complete message
Trigger: customer requests OPSystem
System Reaction: OPSystem list placed orders
Trigger: customer change item number
Guard: item number is not invalid
Trigger: customer submit orders
System Reaction: OPSystem displays success message
Trigger: customer account information is listed
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has not typo
System Reaction: OPSystem display success message
Assertion: Update Order SUCCESS

Figure III-1: Test Scenario from use case: "Place Order" (Update Order SUCCESS I)

Assertion: (OPSystem is ON AND (Customer is logged in OR Sales Clerk is logged in))
Trigger: customer select items
Guard: Item catalog is available
System Reaction: OPSystem mail requested catalog
System Reaction: OPSystem identify items
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is invalid
System Reaction: OPSystem show item number invalid error message
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is not invalid
System Reaction: OPSystem display successful message
Trigger: customer account information is listed
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has typo
System Reaction: OPSystem shows account information error message
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has not typo
System Reaction: OPSystem display success message
Trigger: customer type credit card number
Trigger: customer type expiration date
System Reaction: OPSystem validate credit card information
Guard: Credit card information is valid
System Reaction: OPSystem display payment complete message
Trigger: customer requests OPSystem
System Reaction: OPSystem list placed orders
Trigger: customer change item number
Guard: item number is not invalid
Trigger: customer submit orders
System Reaction: OPSystem displays success message
Trigger: customer account information is listed
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has not typo
System Reaction: OPSystem display success message
Assertion: Update Order SUCCESS

Figure III-2: Test Scenario from use case: "Place Order" (Update Order Success II)

Assertion: (OPSystem is ON AND (Customer is logged in OR Sales Clerk is logged in))
Trigger: customer select items
Guard: Item catalog is not available
System Reaction: OPSystem identify items
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is invalid
System Reaction: OPSystem show item number invalid error message
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is not invalid
System Reaction: OPSystem display successful message
Trigger: customer account information is listed
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has typo
System Reaction: OPSystem shows account information error message
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has not typo
System Reaction: OPSystem display success message
Trigger: customer type credit card number
Trigger: customer type expiration date
System Reaction: OPSystem validate credit card information
Guard: (credit card information is invalid AND customer submit attempts < 4)
Trigger: customer type credit card number
Trigger: customer type expiration date
System Reaction: OPSystem validate credit card information
Guard: Credit card information is valid
System Reaction: OPSystem display payment complete message
Trigger: Sales clerk choose order
System Reaction: OPSystem list orders
Trigger: sales clerk request warehouse send
System Reaction: OPSystem display confirm message
Assertion: Process Order SUCCESS

Figure III-3: Test Scenario from use case: "Place Order" (Process Order Success I)

Assertion: (OPSystem is ON AND (Customer is logged in OR Sales Clerk is logged in))
Trigger: customer select items
Guard: Item catalog is not available
System Reaction: OPSystem identify items
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is not invalid
System Reaction: OPSystem display successful message
Trigger: customer account information is listed
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has typo
System Reaction: OPSystem shows account information error message
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has not typo
System Reaction: OPSystem display success message
Trigger: customer type credit card number
Trigger: customer type expiration date
System Reaction: OPSystem validate credit card information
Guard: Credit card information is valid
System Reaction: OPSystem display payment complete message
Trigger: customer requests OPSystem
System Reaction: OPSystem list placed orders
Trigger: customer change item number
Guard: item number is invalid
System Reaction: order is on hold
Assertion: Update Order FAIL

Figure III-4: Test Scenario from use case: "Place Order" (Update Order FAIL)

Assertion: (OPSystem is ON AND (Customer is logged in OR Sales Clerk is logged in))
Trigger: customer select items
Guard: Item catalog is available
System Reaction: OPSystem mail requested catalog
System Reaction: OPSystem identify items
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is invalid
System Reaction: OPSystem show item number invalid error message
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is not invalid
System Reaction: OPSystem display successful message
Trigger: customer account information is listed
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has typo
System Reaction: OPSystem shows account information error message
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has not typo
System Reaction: OPSystem display success message
Trigger: customer type credit card number
Trigger: customer type expiration date
System Reaction: OPSystem validate credit card information
Guard: (credit card information is invalid AND customer submit attempts < 4)
Trigger: customer type credit card number
Trigger: customer type expiration date
System Reaction: OPSystem validate credit card information
Guard: Credit card information is valid
System Reaction: OPSystem display payment complete message
Trigger: customer select on hold orders
Trigger: customer choose cancel
Trigger: order is cancelled
System Reaction: OPSystem display success message
Assertion: Cancel Order SUCCESS

Figure III-5: Test Scenario from use case: "Place Order" (Cancel Order SUCCESS)

Assertion: (OPSystem is ON AND (Customer is logged in OR Sales Clerk is logged in))
Trigger: customer select items
Guard: Item catalog is not available
System Reaction: OPSystem identify items
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is invalid
System Reaction: OPSystem show item number invalid error message
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is not invalid
System Reaction: OPSystem display successful message
Trigger: customer account information is listed
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has not typo
System Reaction: OPSystem display success message
Trigger: customer type credit card number
Trigger: customer type expiration date
System Reaction: OPSystem validate credit card information
Guard: (credit card information is invalid AND customer submit attempts == 4)
Assertion: Place Order FAIL

Figure III-6: Test Scenario from use case: "Place Order" (Place Order FAIL)

Assertion: (OPSystem is ON AND (Customer is logged in OR Sales Clerk is logged in))
Trigger: customer select items
Guard: Item catalog is available
System Reaction: OPSystem mail requested catalog
System Reaction: *OPSystem identify items*
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is invalid
System Reaction: OPSystem show item number invalid error message
Trigger: customer type item number
System Reaction: OPSystem validate item number
Guard: item number is not invalid
System Reaction: OPSystem display successful message
Trigger: customer account information is listed
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has typo
System Reaction: OPSystem shows account information error message
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has not typo
System Reaction: OPSystem display success message
Trigger: customer type credit card number
Trigger: customer type expiration date
System Reaction: OPSystem validate credit card information
Guard: Credit card information is valid
System Reaction: OPSystem display payment complete message
Trigger: customer requests OPSystem
System Reaction: OPSystem list placed orders
Trigger: customer change item number
Guard: item number is not invalid
Trigger: customer submit orders
System Reaction: OPSystem display success message
Trigger: customer account information is listed
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has typo
System Reaction: OPSystem shows account information error message
Trigger: customer change customer account information
System Reaction: OPSystem verify customer account information
Guard: customer account information has not typo
System Reaction: OPSystem display success message
Assertion: Update Order SUCCESS

Figure III-7: Test Scenario from use case: "Place Order" (Update Order SUCCESS)

Appendix IV: Automatically Generated Test Scenarios (Team Management System) – All Node Coverage

Use-case sequence: "Instructor Login" —*precede*→ "Setup Parameter" —*precede*→ "Visualize Student Teams" —*precede*→ "Add Member to a team" —*precede*→ "Instructor Logout"

Assertion: (TMSystem is ON AND Instructor is NOT logged in)
Trigger: Instructor enters login information
System Reaction: TMSystem checks instructor authorization status
Guard: Instructor is authorized to use TMSystem
System Reaction: TMSystem display instructor operation choices
Trigger: Instructor chooses set up teams parameters
System Reaction: TMSystem asks team parameters.
Trigger: Instructor provides parameters
System Reaction: TMSystem validates parameters
Guard: Team parameters are NOT valid
System Reaction: TMSystem displays team parameters error message
Guard: After 60.0 seconds
System Reaction: TMSystem asks teams parameters
Trigger: Instructor provides parameters
System Reaction: TMSystem validates parameters
Guard: Team parameters are valid
System Reaction: TMSystem displays teams start up acknowledgement message
Guard: After 60.0 seconds
System Reaction: TMSystem displays instructor operation choices
Trigger: Instructor chooces visualize students teams
System Reaction: TMSystem displays list of all teams
Trigger: Instructor selects team
System Reaction: TMSystem displays selected team information
Trigger: Instructor browse select student
Guard: Team status is complete
System Reaction: TMSystem displays team already complete error message
Guard: After 60.0 seconds
System Reaction: TMSystem displays selected team information
Trigger: Instructor browse select student
Guard: Student status is member of team
System Reaction: TMSystem displays student already in team error message
Guard: After 60.0 seconds
System Reaction: TMSystem displays selected team information
Trigger: Instructor browse selected student
Guard: (Team status is NOT complete AND student status is NOT member of team)
System Reaction: TMSystem adds students
System Reaction: TMSystem sends notification added member
System Reaction: TMSystem displays instructor operation choices
Trigger: Instructor Log out
System Reaction: TMSystem display log out acknowledgement
Assertion: Log out SUCCESS

Figure IV-1: Test Scenario from use case: "Instructor Login" (Logout SUCCESS)

Assertion: (TMSystem is ON AND Instructor is NOT logged in)
Trigger: Instructor enters login information
System Reaction: TMSystem checks instructor authorization status
Guard: Instructor is authorized to use TMSystem
System Reaction: TMSystem displays unauthorized access error message
Assertion: Instructor Login FAIL

Figure IV-2: Test Scenario from use case: "Instructor Login" (Login FAIL)

Appendix V: Automatically Generated Test Scenarios (Team Management System) – All-Edge Coverage

Use-case sequence: "Instructor Login" —*precede*→ "Setup Parameter" —*precede*→ "Visualize Student Teams" —*precede*→ "Add Member to a team" —*precede*→ "Instructor Logout"

Assertion: (TMSystem is ON AND Instructor is NOT logged in)
Trigger: Instructor enters login information
System Reaction: TMSystem checks instructor authorization status
Guard: Instructor is authorized to use TMSystem
System Reaction: TMSystem display instructor operation choices
Trigger: Instructor chooses set up teams parameters
System Reaction: TMSystem asks team parameters.
Trigger: Instructor provides parameters
System Reaction: TMSystem validates parameters
Guard: Team parameters are NOT valid
System Reaction: TMSystem displays team parameters error message
Guard: After 60.0 seconds
System Reaction: TMSystem asks teams parameters
Trigger: Instructor provides parameters
System Reaction: TMSystem validates parameters
Guard: Team parameters are valid
System Reaction: TMSystem displays teams start up acknowledgement message
Guard: After 60.0 seconds
System Reaction: TMSystem displays instructor operation choices
Trigger: Instructor chooces visualize students teams
System Reaction: TMSystem displays list of all teams
Trigger: Instructor selects team
System Reaction: TMSystem displays selected team information
Trigger: Instructor browse select student

Guard: (Team status is NOT complete AND student status is NOT member of team)
System Reaction: TMSystem adds students
System Reaction: TMSystem sends notification added member
System Reaction: TMSystem displays instructor operation choices
Trigger: Instructor Log out
System Reaction: TMSystem display log out acknowledgement
Assertion: Log out SUCCESS

Figure V-1: Test Scenario from use case: "Instructor Login" (Instructor Logout SUCCESS)

Assertion: (TMSystem is ON AND Instructor is NOT logged in)
Trigger: Instructor enters login information
System Reaction: TMSystem checks instructor authorization status
Guard: Instructor is authorized to use TMSystem
System Reaction: TMSystem display instructor operation choices
Trigger: Instructor chooses set up teams parameters
System Reaction: TMSystem asks team parameters.
Trigger: Instructor provides parameters
System Reaction: TMSystem validates parameters
Guard: Team parameters are NOT valid
System Reaction: TMSystem displays team parameters error message
Guard: After 60.0 seconds
System Reaction: TMSystem asks teams parameters
Trigger: Instructor provides parameters
System Reaction: TMSystem validates parameters
Guard: Team parameters are valid
System Reaction: TMSystem displays teams start up acknowledgement message
Guard: After 60.0 seconds
System Reaction: TMSystem displays instructor operation choices
Trigger: Instructor chooces visualize students teams
System Reaction: TMSystem displays list of all teams
Trigger: Instructor selects team
System Reaction: TMSystem displays selected team information
Trigger: Instructor browse select student

Guard: Team status is complete
System Reaction: TMSystem displays team already complete error message
Guard: After 60.0 seconds
System Reaction: TMSystem displays selected team information
Trigger: Instructor browse select student

Guard: (Team status is NOT complete AND student status is NOT member of team)
System Reaction: TMSystem adds students
System Reaction: TMSystem sends notification added member
System Reaction: TMSystem displays instructor operation choices
Trigger: Instructor Log out
System Reaction: TMSystem display log out acknowledgement
Assertion: Log out SUCCESS

Figure V-2: Test Scenario from use case: "Instructor Login" (Instructor Logout SUCCESS)

Assertion: (TMSystem is ON AND Instructor is NOT logged in)
Trigger: Instructor enters login information
System Reaction: TMSystem checks instructor authorization status
Guard: Instructor is authorized to use TMSystem
System Reaction: TMSystem display instructor operation choices
Trigger: Instructor chooses set up teams parameters
System Reaction: TMSystem asks team parameters.
Trigger: Instructor provides parameters
System Reaction: TMSystem validates parameters
Guard: Team parameters are NOT valid
System Reaction: TMSystem displays team parameters error message
Guard: After 60.0 seconds
System Reaction: TMSystem asks teams parameters
Trigger: Instructor provides parameters
System Reaction: TMSystem validates parameters
Guard: Team parameters are valid
System Reaction: TMSystem displays teams start up acknowledgement message
Guard: After 60.0 seconds
System Reaction: TMSystem displays instructor operation choices
Trigger: Instructor chooces visualize students teams
System Reaction: TMSystem displays list of all teams
Trigger: Instructor selects team
System Reaction: TMSystem displays selected team information
Trigger: Instructor browse select student

Guard: Student status is member of team
System Reaction: TMSystem displays student already in team error message
Guard: After 60.0 seconds
System Reaction: TMSystem displays selected team information
Trigger: Instructor browse selected student

Guard: (Team status is NOT complete AND student status is NOT member of team)
System Reaction: TMSystem adds students
System Reaction: TMSystem sends notification added member
System Reaction: TMSystem displays instructor operation choices
Trigger: Instructor Log out
System Reaction: TMSystem display log out acknowledgement
Assertion: Log out SUCCESS

Figure V-3: Test Scenario from use case: "Instructor Login" (Instructor Logout SUCCESS)

Assertion: (TMSystem is ON AND Instructor is NOT logged in)
Trigger: Instructor enters login information
System Reaction: TMSystem checks instructor authorization status
Guard: Instructor is authorized to use TMSystem
System Reaction: TMSystem displays unauthorized access error message
Assertion: Instructor Login FAIL

Figure V-4: Test Scenario from use case: "Instructor Login" (Instructor Login FAIL)

Appendix VI: Detail Transition Information of the StateChart in Figure 7-4

State s0_2 Transitions:
0----enter login information/check instructor authorization status---->1

State s1_2 Transitions:
1----[instructor is authorized to use tmsystems]/display instructor operation choices---->2
1----[instructor is NOT authorized to use tmsystem]/display unauthorized access error message---->3

State s0_4 Transitions:
0----choose set up teams parameters/ask teams parameters---->1

State s1_4 Transitions:
1----provide parameters/validate parameters---->2

State s5_4 Transitions:
5----After 60.0 second/ask teams parameters---->1

State s2_4 Transitions:
2----[teams parameters are NOT valid]/display teams parameters error message---->5
2----[teams parameters are valid]/display teams start up acknowledgement message---->3

State s3_4 Transitions
3----After 60.0 second/display instructor operation choices---->4

State s0_3 Transitions:
0----choose visualize students teams/display list of all teams---->1

State s5_1 Transitions:
3----After 60.0 second/display selected team information---->1

State s4_1 Transitions:
4----After 60.0 second/display selected team information---->1

State s0_1 Transitions:
0----select team/display selected team information---->1

State s1_1 Transitions:
1----browse select student/---->2

State s0_0 Transitions:
0----logout/display log out acknowledgement---->1

State s2_1 Transitions:
2----[(team status is NOT complete AND student state is NOT member of team)]/add students, send notification added member, display instructor operation choices---->3
2----[student status is member of team]/display student already in team error message---->5
2----[team status is complete]/display team already complete error message---->4