REFACTORING USE CASE MODELS

KEXING RUI

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE AND SOFTWARE ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

AUGUST 2007

Your file Votre référence
ISBN: 978-0-494-31138-7
Our file Notre référence
ISBN: 978-0-494-31138-7

Abstract

Refactoring Use Case Models

KEXING RUI, Ph.D.
Concordia University, 2007

Use cases are promising vehicles for specifying requirements. However, obtaining well-organized use case models is difficult during software evolution. The thesis proposes to address the issue by refactoring use case models. Refactoring is a program transformation approach for iterative software development. Its concept is introduced to use case models in Cascaded Refactoring.

The thesis introduces major research involved in refactoring use case models. It defines a use case metamodel to formalize use cases. The three-level metamodel covers the environment or context of a use case model, the structure of use cases in terms of episodes, and the event or message passing details of a scenario. The thesis presents a process algebra semantics for the use case model. The episode semantics is provided from the literature. The semantics of a single use case is defined in terms of the episode model. The semantics of the use case model is defined in terms of the individual use cases and their relationships. The thesis identifies a list of properties that need to be preserved during refactoring. It defines fifty-three use case refactorings, which are described using a template covering the refactoring description, arguments, preconditions, postconditions and verification of behavior preservation. The thesis also introduces a tool for use case modeling and refactoring. The tool helps validate the use case metamodel and refactorings on two case studies, which demonstrate that refactoring use case models is feasible and practical. Based on these case studies, the thesis discusses the nature of use case evolution and provides some guidelines for the refactoring process.

To the memory of my mother, for her love, vision and dedication.

Acknowledgments

I would like to thank my supervisor, Dr. Greg Butler, for his support, guidance, patience, kindness, and constant encouragement during my studies. I learned a lot from our regular meetings and from other feedbacks he gave to me.

Many thanks to team members in our research group, Wei Yu, Jian Xu and Renhong Luo, for their fantastic contributions to this research.

I wish to especially thank my wife, LiJuan Liu, for her support through my studies and for loving me whatever I am.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

"The hardest single part of building a software system is deciding what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No part of the work so cripples the resulting systems if done wrong. No other part is more difficult to rectify later" [BRO87]. The Standish Group [STA] has published annual "Chaos" reports on project failure and success factors since 1994 based on the analysis of tens of thousands of projects. According to the group, over 30 percent of the pain factors were strongly related to requirements issues, which include lack of definition and traceability on requirements.

Software requirements engineering is the process of determining what is to be produced in a software system. It identifies stakeholders and their needs, and documents these in an appropriate form for analysis, communication, and subsequent implementation. It is hard to identify real requirements reflecting the verified needs of users for a particular system or capability. The requirements need to be filtered by a process of clarification of their meaning and identification of other aspects that need to be considered. Identifying the real requirements requires an interactive and iterative requirements process, supported by effective practices, processes, mechanisms, methods, techniques, and tools.

Main activities in requirements engineering are [NUS00]:

- eliciting requirements
- modeling and analysing requirements
- communicating requirements
- agreeing requirements
- evolving requirements

The elicitation of requirements is most often regarded as the first step in the requirements engineering process. Requirements are not out there to be collected simply by asking questions. Information gathered during requirements elicitation often has to be interpreted, analysed, modeled and validated. Requirements elicitation needs to find out what problem needs to be solved. It identifies system boundaries. During requirements elicitation, it is critical to identify stakeholders, which include customers or clients, developers, and users. Goals define the objectives a system must meet. In goal-oriented requirements elicitation [DAR93], high-level goals are refined into lower-level goals. It is a continuing activity as development proceeds. Goal-oriented requirements elicitation focuses on the problem domain and the needs of the stakeholders, rather than on possible solutions. Sometimes it is difficult for users to articulate their requirements. A requirement engineer can turn to eliciting tasks users currently perform and those that they might want to perform [JOH92].

Modeling is a fundamental activity in requirements engineering. It constructs abstract descriptions that are amenable to interpretation. Modeling requirements provides the opportunity to analyse them. Modeling approaches are categorized into: enterprise modeling, data modeling, behavioral modeling, domain modeling, and modeling non-functional requirements.

In addition to discovering and specifying requirements, requirements engineering is also a process of facilitating effective communication of these requirements among different stakeholders. There are a variety of formal, semi-formal, and informal languages suggested for requirements documentation [DAV93] [WIE96]. It is crucial to write requirements in a form that is readable and traceable in order to manage their evolution over time. Requirements traceability is the ability to follow the life of a requirement in both forward and backward direction. It provides a rationale for requirements and is the basis for analysing the consequences and impact of change.

As requirements are elicited and modeled, it can be a problem to achieve agreement with all stakeholders. Inspection and formal analysis tend to focus on the coherence of the requirements description. There are two essential difficulties in agreeing and validating requirements. One concerns the question of truth and what is knowable. The other concerns reaching agreement among different stakeholders with conflicting goals. These difficulties are compounded by some contextual issues, including contractual and procurement issues.

Managing change is a fundamental activity in requirements engineering [BOH96]. Successful software systems always evolve. Typical changes to requirements specifications include adding or deleting requirements, and fixing errors. New requirements are added for changing stakeholder needs, or added because they were initially missed. Requirements are deleted to forestall costs and schedule overruns, or satisfy changing stakeholder needs. It is a major challenge to manage inconsistency in requirements specification as they evolve [GHE98]. Managing changing requirements is not just a process of managing documentation, it is also a process of addressing changes through continuing requirements elicitation and risk evaluation. In software engineering, changes on program code lead to a loss of structure and maintainability [BEN00]. Similarly, requirements changes can impact the understandability and maintainability of requirements specification.

Use case modeling is a specific technique for requirements engineering. The use case model is a model that describes the functional behavior of the system in terms of use cases. It consists of all the actors of the system and all the use cases by which actors interact with the system. Ivar Jacobson is regarded as the inventor of use cases. He defines use case as follows [JAC92]: "a use case is a specific way of using the system by using some part of the functionality. A use case constitutes a complete course of interaction that takes place between an actor and the system".

Use cases appeal to the average participant and emphasize the fulfillment of real goals of the end user. They are used to put related requirements together and to describe requirements in a cohesive context. Use cases improve requirements definition and increase user involvement. Now they are widely used in requirement gathering and domain analysis. Use cases are promising vehicles for eliciting, specifying and validating requirements. They provide an effective way to clarify software requirements between the developers and the clients [PRE92] [COC97]. Use case modeling is a very popular way of constructing the requirement model. Use case models are also used to guide scenario-based design and validation and to manage projects [CAR95] [SUT98].

The thesis describes the approach of refactoring use case models to improve their understandability, maintainability, changeability and reusability. Refactoring is a program transformation approach for iterative software development. William Opdyke coins the term "refactoring" to stand for the program restructuring operation that preserves the program behavior for object-oriented applications [OPD92]. Martin Fowler et al. provide guidelines for the refactoring process and explain principles and best practices of program refactorings [FOW99]. An extensive survey of existing research in software refactoring is available in [MENS04].

The concept of refactoring use case models is initially introduced in cascaded refactoring [BUT01]. Lugang Xu proposes the cascaded refactoring methodology [XU06]. Framework development is viewed as framework evolution, which consists of framework refactoring followed by framework extension. A framework is specified by a set of models: feature model, use case model, architectural model, design model, and source code. Framework refactoring is achieved by a set of refactorings cascaded from the feature model, to use case model, architectural model, design model, and source code. Lugang Xu specifies about eight use case refactorings.

The thesis broadens the research in refactoring use case models from the framework development to a more general setting. It describes the approach of applying refactorings to use case models systematically. The purpose of the research is to provide a toolkit for reorganizing requirements. It does not intend to evaluate the quality of the use case model. There is some related work to address the quality issue. Alistair Cockburn illustrates how to write quality use cases [COC01]. Adolph et al. identify about three dozen patterns to evaluate use cases [ADO03]. These patterns provide the observable signs of quality that successful projects tend to exhibit. They can be used as "diagnostics" for judging the quality of use cases. Adolph et al. also indentify some bad smells of the use case model. One of the bad smells is the excessive length of a use case. Another bad smell is that use cases are fragmented and fail to address a complete single goal.

Use case models can be changed by refinement, extension/enrichment, and refactoring. Generally, refinement is the process of deriving an implementation from a specification and verifying the correctness of the derivation. The implementation exhibits those behaviors that are allowed by the specification. Use case models can be refined by introducing more detailed behaviors for use cases. Refined behaviors conforms to those behaviors allowed in the previous use case model. Extension/enrichment is the process of adding new behaviors to use case models. It changes the behavior of the system. Refactoring is the process of changing the structure of use case models without affecting the observable behavior.

The scope of the thesis only covers refactoring. Refactoring makes extension easier but it does not affect the refinement of use case models. Suppose use case model M_1 is a refinement of use case model M_0, M_0' is the use case model after applying refactorings to M_0, and M_1' is the use case model after applying refactorings to M_1. Use case model M_1' must be a refinement of use case model M_0'.

1.2 Overview

Use cases are not formalized. Different people use them differently. The thesis defines a use case metamodel, which formalizes use case syntax and terminology. The metamodel describes related entities that are used to represent the behavior of the use case model. It can be viewed from three different levels: environment level, structure level and event level. At the environment level, a use case is related to external entities. At the structure level, the internal structure of a use case is described in terms of episodes. At the event level, individual events are characterized.

In order to verify behavior preservation of use case refactorings, use case semantics has to be defined. However, there is no standardized use case semantics. The thesis presents a semantics for reasoning behavior preservation using process algebra. Use case semantics is defined as the execution of the use case. The semantics of the use case model is defined as the set of possible executions of all use cases within the model. Hence concurrency is not concerned in the thesis. Since an episode can be described using a Basic Message Sequence Chart [AND97] and the semantics of a Basic Message Sequence Chart can be defined using process algebra [MAUW94], we choose process algebra to define use case semantics for convenience. We do not need the full power of process algebra to specify the set of event traces in the use case model. The thesis uses process algebras PA_{BMSC} [MAUW94] to define the episode semantics. We summarize PA_{BMSC} in Section 2.5. Process algebra provides a natural mapping for the sequence of events in the use case model. Relationships between events are mapped into binary operators in process algebra directly. The semantics of a use case is defined in terms of the process algebra semantics of the episode model. Each use case corresponds to a process algebra term which defines what happens when the use case is executed.

The semantics of the use case model is defined as the alternation of the process algebra terms of all use cases which can be executed.

The thesis identifies eight invariants required to preserve the behavior of the use case model. They represent properties that have to be preserved during refactorings. These invariants ensure that the use case model after refactorings is syntactically correct and semantically equivalent.

Based on the use case metamodel, the use case semantics, and these invariants, fifty-three use case refactorings are defined in detail. A template is designed to specify each use case refactoring in terms of the refactoring description, arguments, preconditions, postconditions and verification of behavior preservation.

In order to evaluate these refactorings, a use case modeling and refactoring tool is developed. Three master students join my research group to implement the tool: Jian Xu [XU04], Wei Yu [YU04] and Renhong Luo. Furthermore, Jian Xu and Wei Yu carry out some case studies to validate use case refactorings with the tool support. I lead the group and guide their work.

The tool can be used to build the use case model based on the above use case metamodel. The model can be evolved using use case refactorings with the tool support. This helps to verify the use case metamodel and refactorings. Two case studies are introduced in the thesis, which validate the tool and refactorings. It is demonstrated that refactoring use case models is feasible and practical. Finally, the thesis analyses the refactoring process. It discusses the nature of use case evolution and provides some guidelines on the refactoring process.

The rest of the thesis is organized as follows.

Chapter 2 introduces related background information. This work is originated from the source code refactoring. This chapter starts by illustrating the refactoring concept and its major milestones. It introduces background information on use case modeling. It summarizes some related work on model transformations. Since use case refactorings need to preserve the behavior of the use case model, it raises the question of how to define the semantics of the model. This chapter surveys current approaches used to define semantics for Message Sequence Charts (MSCs), statecharts, activity diagrams, sequence diagrams and use cases. It also summarizes the approach of defining the semantics of Basic Message Sequence Charts using process algebra PA_{BMSC} [MAUW94] because the process algebra semantics defined in Chapter 3 for the use case model is based on this approach.

Chapter 3 describes in detail our work on refactoring use case models. A three-level use case metamodel is defined to formalize use cases. It covers the environment or context of a use case model, the structure of use cases, and the event or message passing details of a scenario. This metamodel supports requirements engineering and software engineering. In order to verify the behavior preserving property of the individual use case refactoring, a process algebra semantics is introduced to define the use case model. A list of invariant properties of the model are identified in order for refactorings to preserve the behavior of the use case model. This chapter also presents a template to describe use case refactorings. It gives an overview of the use case refactorings. A brief description is given for some refactorings. The detailed definition of all refactorings is presented in Appendix.

Chapter 4 describes the implementation of a use case modeling and refactoring tool. The tool is developed based on DrawLets, which is a two-dimensional graphics framework for building graphics applications. This chapter introduces the Definition subsystem and Refactoring subsystem, and describes data structures and storage, which are based on the three-level metamodel. It illustrates four major modules: use case tool, goal tool, episode tool and event tool.

Chapter 5 presents two case studies for validating the tool and use case refactorings. One is the Automatic Teller Machine (ATM) and the other is the Video Store System (VSS). These case studies show the power of refactorings during the use case evolution. They are analysed in terms of three aspects: evaluation of the tool, validation of use case refactorings, and the refactoring process. These are illustrated within evaluation conclusions of this chapter.

Chapter 6 concludes the thesis by answering some important research questions. It describes contributions of this research by comparing with some related work. This chapter also discusses limitations and future work.

1.3 Contributions

Major contributions made in this research are:

1. A three-level use case metamodel with a hierarchical structure.

   This metamodel represents a new perspective in use case formalization. It covers the environment or context of a use case model, the structure of use cases in terms of episodes, and the event or message passing details of a scenario. The metamodel supports requirements engineering and software engineering. It defines different use case relationships and the actor relationship. It introduces the concept of episode model and episode tree, which describe not only primitive episodes but also their relationships. Similarly, it also introduces the event model and event tree.

2. A process algebra semantics for the use case model.

   The semantics of a single episode is specified by a process algebra PA_{BMSC}. The semantics of a single use case is specified through the episode model. Then the semantics of a use case model is defined based on use case relationships and the episode model of each use case. This process algebra semantics makes it possible to verify the behavior preservation of the use case refactoring.

3. A list of fifty-three use case refactorings.

   The thesis defines fifty-three refactorings using a template. Each refactoring is described in terms of the refactoring description, arguments, preconditions, postconditions, and the verification of the behavior preservation. These refactorings provide various transformation capabilities including generalization of use cases and actors, decomposition of use cases and episodes, and composition of use case relationships and episode relationships. They are very useful and effective for reorganizing use case models.

4. A use case modeling and refactoring tool.

   This tool provides an integrated environment for creating a use case model based on the above use case metamodel and for refactoring this use case model. It facilitates the refactoring process. It allows to check refactoring preconditions and to apply the refactoring automatically.

5. Validation of the use case refactorings with the tool support using two case studies.

   These case studies demonstrate that it is feasible and practical to evolve the use case model starting with a simple model and to improve the structure of the more complex use case model by refactorings. The thesis investigates the evolution process of use case models. It provides guidelines on the refactoring process.

Chapter 2

Related Work

This chapter introduces related work on source code refactoring, requirements engineering with use cases, and model transformations. It also surveys current approaches used to define the semantics for Message Sequence Charts (MSCs), statecharts, activity diagrams, sequence diagrams and use cases. It reviews process algebras PA_{ε} and PA_{BMSC} used to define the semantics of Basic Message Sequence Charts in detail because they are used to define the process algebra semantics for the use case model in Chapter 3.

2.1 Source Code Refactoring

A refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify, without changing the observable behavior of that software component [FOW99].

William F. Opdyke's PhD thesis [OPD92] is the first serious publication in refactorings. Inspired by the work of Banerjee and Kim for object-oriented database schema evolutions [BAN87], he focuses on automatic support for program restructuring (refactoring) to the object-oriented program. In his PhD thesis, he identifies seven invariants required to preserve the behavior of C++ programs. When refactoring may violate an invariant, enabling conditions are added to ensure that the invariant is preserved. The seven invariants are as follows:

- Unique Superclass: It supports only single inheritance systems, without cycles in the inheritance graph. A class must always have at most one direct superclass and its superclass must not also be one of its subclasses.
- Distinct Class Names: Each class must have a unique name. The scope of each class is the entire program.

- Distinct Member Names: All member variables and functions within a class must have distinct names. It allows a member function in a superclass to be overridden in a subclass.
- Inherited Member Variables Not Redefined: A member variable inherited from a superclass cannot be redefined in any of its subclasses.
- Compatible Signatures in Member Function Redefinition: If a member function defined in a superclass is redefined in a subclass, this function must be virtual so that it can be overridden in subclasses. The signature of this function must be compatible with the one in subclasses.
- Type-Safe Assignments: The type of each expression assigned to a variable must be an instance of the variable's defined type or an instance of one of its subtypes.
- Semantically Equivalent References and Operations: The versions of the program before and after a refactoring must produce semantically equivalent references and operations. Semantic equivalence is defined as: the resulting set of output values must be the same if the program is called twice (once before and once after a refactoring) with the same set of inputs. This allows for several important changes that do not affect equivalence:
  - Expressions can be simplified and dead code removed.
  - Variables, functions and classes can be added if they are unreferenced.
  - The type of a variable can be changed by a refactoring, as long as each operation referenced on the variable is defined equivalently for its new type.
  - References to a variable and function defined in one class can be replaced by references to an equivalent variable or function defined in another class.

In his thesis he defines twenty-six low level refactorings. They are grouped into five categories. These refactorings are listed in Table 1.

Each refactoring has some preconditions. Because these small refactorings are correct under certain preconditions, large changes that are composed solely of small refactorings must be correct. Therefore, refactoring can support software design and evolution by restructuring a program in a way that allows other changes to be made more easily. Complicated changes to a program can require both refactorings and additions.

With these low level refactorings, he defines three high level refactorings, which are more abstract. These high level refactorings are:

- Refactoring to generalize: creating an abstract superclass
- Refactoring to specialize: subclassing, and simplifying conditionals

Table 1: Source Code Refactorings

Category	**Refactoring Name**
Creating a program entity	(a) create_empty_class (b) create_member_variable (c) create_member_function
Deleting a program entity	(a) delete_unreferenced_class (b) delete_unreferenced_variable (c) delete_member_functions
Changing a program entity	(a) change_class_name (b) change_variable_name (c) change_member_function_name (d) change_type (e) change_access_control_mode (f) add_function_argument (g) delete_function_argument (h) reorder_function_arguments (i) add_function_body (j) delete_function_body (k) convert_instance_ variable_to pointer (l) convert_variable_references_ to_function_calls (m) replace_statement_list_ with_function_call (n) inline_function_call (o) change_superclass
Moving a member variable	(a) move_member_variable_ to_superclasses (b) move_member_variable_ to_subclasses
Intermediate level (composite) refactorings	(a) abstract_access_to_ member_variable (b) convert_code_segment_ to_function (c) move_class

- Refactoring to capture aggregation and components

Based on Opdyke's research, Lance Tokuda goes further to evolve object-oriented designs with refactorings [TOK99]. His research shows that all three kinds of design evolution, which are schema transformations, design pattern microarchitectures, and the hot-spot-driven-approach, are automatable with refactorings.

Schema transformations perform many of the simple edits encountered when evolving class diagrams. They can be used alone or in combination to evolve object-oriented designs. The schema for an object-oriented database management system (OODBMS) looks like a class diagram of an object-oriented application. Among nineteen object-oriented database schema transformations, twelve transformations were implemented as automated refactorings by Tokuda.

As with database schema transformations, he shows refactorings to directly implement certain design patterns from [GAM95]. Six patterns are automatable as refactorings: Command, Composite, Decorator, Factory Method, Iterator and Singleton.

A number of patterns can be viewed as automatable program transformations applied to an evolving design. At least seven patterns from [GAM95] can be viewed as a program transformation, which are: Abstract Factory, Adapter, Bridge, Builder, Strategy, Template Method and Visitor.

The hot-spot-driven-approach provides a comprehensive method for evolving designs to manage changes in both data and functionality. Meta patterns can be added to evolve designs. Most of them can be viewed as transformations from a simpler design. Refactorings automate the transition between designs granting designers the freedom to create simple frameworks and add patterns as needed when hot-spots are identified.

To make refactoring more practical, Donald Bradley Roberts develops a commercial-grade tool, the Refactoring Browser for smalltalk. In his PhD thesis [DON99], he illustrates ways to make a refactoring tool both fast and reliable so that it is more useful.

First of all, he extends the definition of refactoring presented by William F. Opdyke by adding postconditions, which are assertions that a program must satisfy for the refactoring to be applied. Postconditions describe how the assertions are transformed by the refactoring and they can be used for several purposes: to reduce the amount of analysis that later refactorings must perform, to derive preconditions of composite refactorings, and to calculate dependencies between refactorings. These techniques can be used in a refactoring tool to support undo, user-defined composite refactorings, and multi-user refactoring.

He also defines a method to calculate the preconditions for composite refactorings. Each atomic refactoring within the composite has its own preconditions that must be true for the refactoring to be legal. Given a sequence of refactorings that make up a composite refactoring, it would be nice to be able to derive the preconditions for the composite refactoring from the individual refactorings that it comprises.

In addition, he defines the dependency between refactorings based on commutativity. Large design changes can be composed of a sequence of smaller, more primitive refactorings. Since each step is a refactoring, the entire composition is also a refactoring. This composition property is very powerful in that it allows us to define and automate simpler, low-level refactorings, and then compose them into larger, complex refactorings. However, although the refactorings were initially specified in a sequence, they do not necessarily have to be performed in that sequence. Therefore, it is important to determine which refactorings must occur before other refactorings. Donald Roberts defines the formula for calculating the conditions under which any two refactorings may commute.

Further, he implements a set of applications that use the dependency information calculated from a chain of refactorings. Three of these applications are as follows:

- Undo: Undoing a refactoring in the middle of the chain of refactorings requires undoing some refactorings that were performed later in the chain. He uses the dependency information to determine which refactorings must be undone in the chain.
- Parallelizing Refactorings: Given a chain of refactorings, he uses the dependency information to determine which sets of refactorings within the chain can be performed in parallel, and which ones must be performed sequentially.
- Merging Refactorings in Multiuser Programming Environments: He uses the dependency information to determine if two chains of refactorings created by different programmers conflict. If two chains of refactorings do conflict, he uses the dependency information to find subsequences from the front of each chain that do not conflict with each other.

Finally, he develops a scheme for using dynamically obtained information to perform refactorings, which can eliminate expensive static analysis by deferring the analysis to runtime.

Compared with Donald Roberts, Martin Fowler focuses on another direction, the refactoring process, to make refactoring more practical. With contributions of Kent Beck, John Brant, William F. Opdyke, and Donald Roberts, Martin Fowler publishes a book [FOW99], *Refactoring: Improving the Design of Existing Code*, which explains the principles and best practices of refactorings. It also provides a guideline for the refactoring process: where to start refactorings, when to start, when to stop. The core of the book is a comprehensive catalog of refactorings. Each refactoring illustrates the motivation and mechanics of a proven code transformation.

Extreme Programming(XP) [BECK99] is a methodology developed by Kent Beck whose central idea is that you work on one use case at a time and only design the software to handle the use case that you are working on. If a particular use case does not fit well into the design, refactor the design until the use case can be implemented in a reasonable manner. One of the key aspects of XP is continual and aggressive refactoring. XP increases the visibility of refactoring source code within the community.

Currently there are many refactoring tools available for some popular languages. For example, Eclipse [ECLIPSE] is a Java IDE that provides strong support to refactoring Java programs. JRefactory [JREF] is a plug-in for JBuilder, NetBeans, and Elixir IDEs. JBuilder [BORL] and JDeveloper [ORAC] support refactorings on Java programs. Refactor! Pro [DEVE] is a general .NET refactoring tool that supports both C# and Visual Basic programs. Ref++ [IDEA] is a visual studio add-in that provides refactoring support for C++ programs.

2.2 Requirements Engineering with Use Cases

Use cases are promising vehicles for eliciting, specifying and validating requirements. Ivar Jacobson is regarded as the inventor of use cases. He defines use case as follows [JAC92]: "a use case is a specific way of using the system by using some part of the functionality. A use case constitutes a complete course of interaction that takes place between an actor and the system".

Although this definition is brief, broad, and imprecise, his introduction of use cases immediately improves the situation that requirement documents often have a poor fit with both business reengineering and implementation. Use cases are now widely used in requirement gathering and domain analysis. Currently, there are various approaches to describe and formalize use cases. They represent different perspectives on use case modeling.

Jacobson, Griss, and Jonsson provide a thorough methodology addressing architectural, process, and organizational aspects of software reuse [JAC94]. They propose requirement gathering through use scenarios, first captured informally, then expressed more formally in a use case model. The use case model is considered as the starting point for the test model. Each execution of a use case described as a scenario will correspond to one test case. They pay considerable attention to component systems and variability mechanisms. Variability in component systems occurs at variation points and utilizes one of three mechanisms: inheritance, configuration, and parameterization. Jacobson et al. introduce an approach toward improving use case modeling by fortifying it with aspect orientation [JAC05].

Cockburn proposes to structure use cases with goals [COC97]. He identifies four dimensions to use case descriptions: purpose, content, plurality, and structure. Each of these dimensions has an enumerated domain value. Purpose can be either for user stories or requirements. Content can be either contradicting, consistent prose, or formal content. Plurality is either 1 or multiple. Structure can be unstructured, semi-formal, or formal structure. He introduces a theory based on a small model of communication, distinguishing "goals" as a key element of use cases. His approach is defined as requirements, consistent prose, multiple scenario, and semi formal structure, which is the same as Jacobson's approach. Cockburn provides guidelines on how to write quality use cases [COC01]. Complementary to Cockburn's work, Adolph et al. provide guidelines and patterns to evaluate use cases [ADO03].

Buhr develops Use Case Maps (UCMs) as a visual notation for comprehending and developing the architecture for emergent behavior in large, complex, self modifying systems [BUHR98]. UCMs are a two dimensional map of cause-effect chains from points of stimuli through the system to points where responses are observed. UCMs consist of three primary constructs. Responsibilities are represented as dots, with the responsibility described by active natural language phrases. Causality is represented as a path that connects the dots, with start and end points that have associated pre- and post-conditions. Components are represented as simple boxes with associated responsibilities. UCMs can be refined through decomposition and partitioned by factoring.

The Unified Modeling Language (UML) [UML20] is a standardized specification language to model object-oriented systems. There are three prominent parts of a system's model: functional model, object model, and dynamic model. The functional model showcases the functionality of the system from the user's point of view. It includes *use case diagrams*. The object model showcases the structure and substructure of the system using objects, attributes, operations, and relationships. It includes *class diagrams*. The dynamic model showcases the internal behavior of the system. It includes *sequence diagrams*, *activity diagrams*, and *state machine diagrams*. UML's definition for use case has shifted from Jacobson's original emphasis on use to a more "system-centric" viewpoint. According to UML: "a use case is the specification of sequences of actions, including variant sequences and error sequences, that a system, subsystem, or class can perform by interacting with outside actors".

The Rational Unified Process (RUP) [KRU03] is a guide for how to effectively use the UML. The RUP is a software engineering process. It provides a disciplined approach to assigning tasks and responsibilities within a development organization. It enhances team productivity by providing every team member with easy access to a knowledge base with guidelines, templates and tool mentors for all critical development activities. The knowledge base allows development teams to gain the full benefits of the UML. The RUP is supported by tools, which automate large parts of the process.

The OPEN Modeling Language (OML) [FIR98] is a competing notation to the UML. It represents the merger of three methodologies: SOMA [SEL97], MOSES [SEL93], and Firesmith [SELL97]. A key principle behind OML is the notion of tasks and techniques. A task may be accomplished by one or more appropriate techniques. In OML, the relationship between a use case and a scenario is described in several ways as a specialization of a use case, an instance of a use case, and as a component of a use case. A use case links with objects via a participation association. OML also defines a specific category of scenario relationships, which are precedes, invokes and uses. In OML, capturing user requirements involves the use of task scripts, which are supported by a task-action grammar that consists of Subject - Verb - Direct Object - Preposition - Indirect Object (SVDPI). These task scripts can be organized into composition, classification and usage structures.

Regnell investigates the role of use case modeling in requirements engineering and its relation to system verification and validation [REG99]. As shown in Figure 1, his approach utilizes three levels for use case modeling. It allows a hierarchical structure and enables graphical representation at different abstraction levels. At the environment level, the use case is related to the entities external to intended system. At the structure level, the internal structure of a use case is revealed together with its different variants and parts. The event level represents a lower abstraction level where the individual events are characterized. He introduces the process of Usage-Oriented Requirements Engineering, which is an extension of use case driven analysis, and the Synthesized Usage Model resulting from this process. He investigates the possibility of integrating the two disciplines of use case modeling and statistical usage testing and discusses how they can be integrated to form a seamless transition from requirements models to test models for reliability certification. Two approaches, which are transformation and extension, are proposed for the integration of the use case model and the operational profile model.

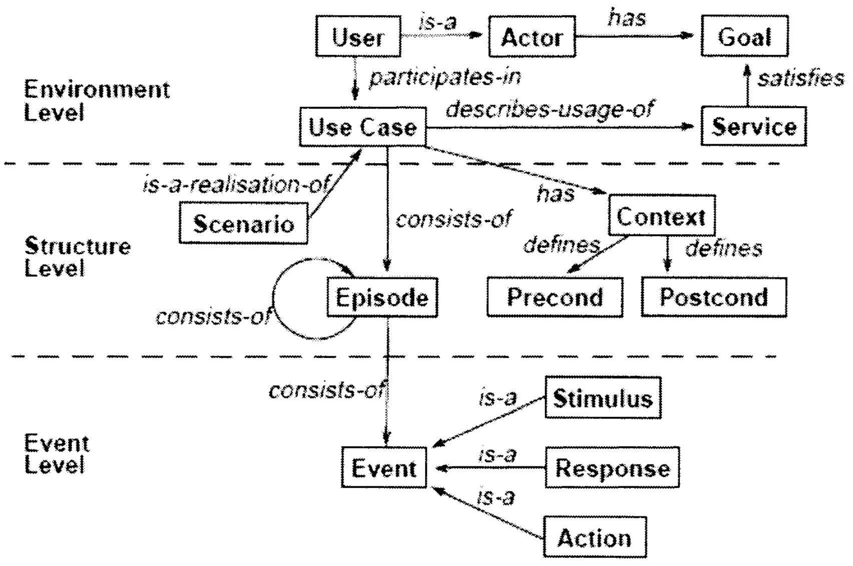


Figure 1: Concept Relations and Levels of Abstraction [REG99]

2.3 Model Transformations

Refactoring use case models is an example of model transformations. This section introduces some related work on model transformations. It helps understand the context of refactoring by investigating current approaches used in use case models as well as other models.

Marco Boger et al. illustrate how refactorings can be directly defined on the level of models rather than on code [BOG02]. They discuss which refactoring fits better to a UML representation. They describe static structure refactorings, state machine refactorings and activity graph refactorings. In static structure refactorings, some source code refactorings can be applied to class diagrams directly. Since UML is a two-dimensional and graphical representation, consequences of refactorings can be overviewed better. Some source code refactorings, like the replacement of inheritance by delegation or the extraction of a common interface from a set of classes, are more apparent on a model level. In state machine refactorings, they introduce *Merge States* refactoring, *Decompose Sequential, Composite State* refactoring, *Form Composite State* refactoring and *Sequentialize Concurrent, Composite State* refactoring. In activity graph refactorings, they introduce *Make Actions Concurrent* refactoring and *Sequentialize Concurrent Actions* refactoring. They focus on the automatic detection of conflicts. Conflicts are grouped into warnings and errors. While warnings indicate that a refactoring might cause a side effect, errors show damages caused to the model. They illustrate how these refactorings are implemented in a refactoring browser.

Gerson Sunyé et al. [SUNY01] define refactorings for UML class diagrams and statecharts. Refactorings on class diagrams are summarized in five basic operations: *addition*, *removal*, *move*, *generalization* and *specialization* of modeling elements. These operations are defined as follows.

- The *Addition* of features (attributes and methods) and associations to a class can be done when the new feature (or association) does not have the same signature as any other features owned by the class or by its parents.
- The *Removal* of associations and features can only be done when these elements are not referenced in the whole model.
- The *Move* is used to transfer a method from a class to another, and create a forwarder method in the former.
- The *Generalization* refactoring can be applied to elements owned by classes, such as attributes, methods, operations, association ends and statecharts. It consists in the integration of two or more elements into a single one which is transfered to a common superclass.
- The *Specialization* refactoring is the exact opposite of *Generalization*. It consists in sending an element to all direct subclasses of its owner.

In the case of statecharts, they introduce the following refactorings: *Fold Incoming / Outgoing Actions*, *Unfold Entry/Exit Action*, *Group States*, *Fold Outgoing Transitions*, *Unfold Outgoing Transition*, *Move State into Composite*, *Move State out of Composite*.

They formalize refactorings using the Object Constraint Language (OCL) [UML99] at the meta-model level to specify behavior-preserving transformations. However, since the abstract syntax of the OCL is not precisely specified, they are not able to analyse the contents of a constraint and use this information to improve the definition of some OCL-based refactorings. They are unable to formalize the relation between the class (structural) and statechart (behavioral) diagrams.

Robert France et al. propose a metamodeling approach to pattern-based model refactoring [FRA03]. They develop metamodels called transformation specification that characterize families of transformations. These metamodels are used to check transformations for conformance. The design models are expressed in the UML. They define a pattern specification which includes *problem specification*, *solution specification* and *transformation specification*. *Problem specification* is a precise specification of the family of UML design problems that the pattern addresses. *Solution specification* is a precise specification of the UML designs representing solutions of the pattern. *Transformation specification* is a specification of problem-to-solution transformations. The whole specifications can be viewed as defining UML metamodel specializations. To apply refactorings to a design model using a selected design pattern, the design model needs to be checked against the pattern *problem specification* to determine if the chosen design pattern can be applied to the source model. If the condition is satisfied, the transformation steps can be developed using the transformation language defined by the *transformation specification* at the metamodel level. A target model can be produced by applying the transformations to the source model.

They also provide tool support to automate the process of applying pattern-based transformations. This helps to reduce the effort of consistently and correctly realizing patterns across a design. The tool provides two interfaces. One is for evolving and manipulating the UML metamodel. The other is for creating, manipulating and evolving UML models using patterns.

Paolo Bottoni et al. describe software refactorings by distributed graph transformation [BOT03]. Source code and diagrams are abstracted to graphs. They use explicit flow graphs for source code and UML metamodel for diagrams. Overlapping in common elements is expressed by interface graphs. They propose to coordinate refactorings over different UML diagrams by hierarchical distributed graph transformation. They use the following UML diagrams: class diagrams, sequence diagrams, and state machine diagrams.

Pieter Van Gorp et al. investigate ways to maintain the consistency between the model and the code while one of them gets refactored [GORP03]. They introduce a list of design criteria that should be met by any UML version to support refactorings. These criteria are summarized as follows.

- Integration with Code Smell Detectors: *Code smells* indicates problematic code fragments that should be refactored. In order to automatically suggest refactorings based on a set of detected code smells, the metamodel should be adequate for describing such code smells and exchanging them with a detector.

- Consistency Maintenance between Model and Code: Source-consistent refactoring on UML models is only feasible if the UML metamodel is sufficiently expressive to capture the effects of refactorings on the underlying sources.

- Minimal Additions: The UML specification describes two extension mechanisms, one being the "lightweight" profile mechanism, the other being the "heavyweight" Meta-Object Facility

(MOF) [UML99] approach. The profile mechanism is restricted to adding attributes and constraints to existing model elements, whereas MOF allows the addition of new model elements. The more lightweight the extension, the easier it is for existing tools to accommodate them into their repository, hence it is better to keep the extension as minimal as possible.

- Maximal Effect: By careful selection of the places where the metamodel is extended, it is possible that a minimal extension has maximal effect.

- Backward Compatibility: When extending or reusing an existing model element, changes to the semantics should be avoided. Such conservative extensions guarantee that all existing UML models remain valid.

- For all Common Object-Oriented Languages: One of the key benefits of the UML metamodel is that it abstracts away language syntax without loosing the basic Object-Oriented constructs (classes, methods and attributes, etc.). When extending this core, the balance should be kept between achieving enough precision for refactoring and maintaining language independence.

They argue that the UML 1.4 metamodel is inadequate for maintaining the consistency between a refactored design model and the corresponding program code. They propose a UML 1.4 extension called *GrammyUML* using OCL expressions. *GrammyUML* consists of heavyweight extensions, lightweight extensions and Well-Formedness Rules. In heavyweight extensions, they propose eight additive and language-independent extensions to the UML 1.4 metamodel which form the foundation of GrammyUML. These extensions are listed as follows:

1. Relate a *Method* to its contained statements, i.e. to *ActionSequence.*

2. Add *LocalVariable* as a specialization of *ModelElement.*

3. Relate a *LocalVariable* to its type, i.e. to *Classifier.*

4. Relate a *LocalVariable* to its surrounding scope, i.e. to *ActionSequence.*

5. Relate an *Action* to its actual arguments, i.e. to *Argument.*

6. Refine the *value* attribute of *Argument* to the *valueRefinement* association-end of type *ModelElement.*

7. Add *SingleTargetAction* as a specialization of *Action.*

8. Refine the *target* attribute of *SingleTargetAction* to the *targetRefinement* association end of type *ModelElement.*

They introduce the concept of *refactoring contract*, which consists of three sets of constraints per refactoring: precondition, postcondition and the code smells. They illustrate how refactoring pre- and post-conditions can be specified in OCL using two sample refactorings: *Pull Up Method* and *Extract Method*. They describe how OCL refactoring contracts can be used to compose primitive refactorings and how *GrammyUML* enables them to collect metrics data from inside the method body and describe the code smells.

John McGregor et al. develop *use case assortment* for the requirement analysis phase of framework development [MIL99]. This approach combines a set of modeling heuristics with an analysis technique to identify commonality and variability among use cases within the use case model. To reflect the commonality/variability analysis, the use case model is factored by introducing abstract use cases and abstract actors, and rearranging responsibilities.

Inspired by the *use case assortment* approach, the concept of refactoring use case models is proposed in *Cascaded Refactoring* [BUT01] [XU06]. *Cascaded Refactoring* is a hybrid approach for the development and evolution of application frameworks. It combines the modeling aspects of top-down domain engineering approaches and the iterative, refactoring approaches of the bottom-up object-oriented community. The set of models used in *Cascaded Refactoring* are:

- a feature model organizing common and variable features;
- a use case model of requirements including variation points;
- an architectural design with hotspots;
- a design showing collaborations and the overlapping of roles from design patterns onto the hotspots; and
- the source code, with classes, hooks, and templates.

In *Cascaded Refactoring*, refactoring is viewed as an issue-driven activity. The restructuring of one model determines constraints on the restructuring of other models via the traceability and trace maps. The trace maps are:

- the trace map from the capability feature model to the use case model;
- the trace map from the operating environment feature model to the architectural model;
- the trace map from the domain technology feature model to the design model;
- the trace map from the implementation feature model to the source code;
- the trace map from the use case model to the architectural model;

- the trace map from the use case model to the design model;
- the trace map from the architectural model to the design model; and
- the trace map from the design model to the source code.

Although some use case refactorings are given in *Cascaded Refactoring*, detailed refactorings are not investigated within this approach.

2.4 Model Semantics

In order to refactor use case models, the semantics of use cases has to be defined. This section investigates current approaches used to define semantics for different models, not necessarily for use case models. It helps understand and justify the use case semantics defined in Section 3.2.1.

Current approaches on model semantics are mostly based on automaton theory, process algebra and Petri nets. This section starts by introducing the background information on these underlying theories. Then it describes the related work on defining the semantics for Message Sequence Charts (MSCs), statecharts, activity diagrams, sequence diagrams and use cases. Finally it introduces a process algebra for Basic Message Sequence Charts in detail as this is a basis for the use case semantics in the thesis.

2.4.1 Background on Underlying Theories

This section surveys automaton theory, process algebra and Petri nets respectively. It describes the background information on automaton theory and process algebra in detail while briefing Petri nets. The use case semantics in the thesis is based on process algebra.

Automaton Theory

An automaton is a mathematical model for a finite state machine [HOP01]. A finite state machine is a machine that proceeds through a series of states according to a transition function (can be seen as a table) that tells the automaton which state to go to next given a current state and a current input symbol. The input is read symbol by symbol, until it is consumed completely (think of it as a tape with a word written on it, that is read by a reading head of the automaton; the head moves forward over the tape, reading one symbol at a time). Once the input is depleted, the automaton is said to have stopped. Depending on the state in which the automaton stops, it is said that the automaton accepts or not the input. If it landed in an accept state, then the automaton accepts the word. If, on the other hand, it lands on a non-accept state, the word is rejected. The set of all the words accepted by an automaton is called the language accepted by the automaton.

Turing machines are the most general automata. They consist of a finite set of states and an infinite tape which contains the input and is used to read and write symbols during the computation. Since Turing machines can leave symbols on their tape at the end of the computation, they can be viewed as computing functions: the partial recursive functions. Despite the simplicity of these automata, any algorithm that can be implemented on a computer can be modeled by some Turing machine.

The behavior of a system is the total of events or actions that a system can perform, the order in which they can be executed and maybe other aspects of this execution such as timing or probabilities. Always, we describe certain aspects of behavior, disregarding other aspects, so we are considering an abstraction or idealization of the real behavior. Rather, we can say that we have an observation of behavior, and an action is the chosen unit of observation. The simplest model of behavior is to see behavior as an input/output function. This model is instrumental in the development of (finite state) automata theory. In automata theory, a process is modeled as an automaton. An automaton has a number of states and a number of transitions, going from one state to another state. A transition denotes the execution of an (elementary) action, the basic unit of behavior. Besides, there is an initial state (sometimes, more than one) and a number of final states. A behavior is a run, i.e. a path from initial state to final state.

Process Algebra

The term process algebra is used in different meanings. J. C. M. Baeten surveys the history of process algebra and gives the following definition: "process algebra is the study of pertinent equational theories with their models, while the wider field that also includes the study of transition systems and related structures, ways to define them and equivalences on them will be called process theory" [BAE04].

The history of process algebra can be traced back to the early seventies. In 1970, we can distinguish three main styles of formal reasoning about computer programs, focusing on giving semantics (meaning) to programming languages.

- Operational Semantics: A computer program is modeled as an execution of an abstract machine. A state of such a machine is a valuation of variables, a transition between states is an elementary program instruction. Pioneer of this field is McCarthy [MCC63].

- Denotational Semantics: More abstract than operational semantics, computer programs are usually modeled by a function transforming input into output. Most well-known are Scott and Strachey [SCO71].

- Axiomatic Semantics: Here, emphasis is put on proof methods proving programs correct. Central notions are program assertions, proof triples consisting of precondition, program statement and postcondition, and invariants. Pioneers are Floyd [FLO67] and Hoare [HOA69].

However, it turns out that it is difficult to give semantics to programs containing a parallel operator using the methods of denotational, operational or axiomatic semantics. There are two paradigm shifts that need to be made before a theory of parallel programs in terms of a process algebra can be developed. Firstly, the idea of a behavior as an input/output function needed to be abandoned. A program could still be modeled as an automaton, but the notion of language equivalence is no longer appropriate. This is because the interaction a process has between input and output influences the outcome, disrupting functional behavior. Secondly, the notion of global variables needs to be overcome. Using global variables, a state of a modeling automaton is given as a valuation of the program variables, that is, a state is determined by the values of the variables. The independent execution of parallel processes makes it difficult or impossible to determine the values of global variables at a given moment.

Bekič contributes basic ingredients to the emergence of process algebra [BEK71] [BEK84]. He defines a denotational semantics for parallel composition.

Robin Milner is the central person in the history of process algebra. He develops his process theory CCS (Calculus of Communicating Systems) [MIL80]. He formulates the semantics of parallel composition within the framework of denotational semantics using *transducers*. He considers the problems caused by non-terminating programs, with side effects, and non-determinism. He defines operations for sequential composition, alternative composition and parallel composition. He presents the equational laws as truths about his chosen semantical domain, rather than considering the laws as primary, and investigating the range of models that they have. David Park contributes it with the formulation of bisimulation [PARK81].

Tony Hoare is a very important contributor to the development of process algebra. He defines the language CSP (Communicating Sequential Processes) [HOA78]. He completely gets rid of global variables and adopts the message passing paradigm of communication. His paper [HOA78] inspired Milner to treat message passing in CCS in the same way.

Jan Bergstra and Jan Willem Klop use the phrase of "process algebra" for the first time [BER82]. It is quoted here:

A *process algebra* over a set of atomic actions A is a structure $\mathcal{A} = \langle \mathrm{A}, +, \cdot, \lfloor\!\lfloor, a_i(i \in I) \rangle$ where A is a set containing A, the a_i are constant symbols corresponding to the $a_i \in A$, and + (*union*), (*concatenation* or *composition*, left out in the axioms), $\lfloor\!\lfloor$ (*left merge*) satisfy for all x, y, z ∈ A and a ∈ A the following axioms:

PA1 $x + y = y + x$

PA2 $x + (y + z) = (x + y) + z$

PA3 $x + x = x$

PA4 $(xy)z = x(yz)$

PA5 $(x + y)z = xz + yz$

PA6 $(x + y) \mathbin{\|\!\lfloor} z = x \mathbin{\|\!\lfloor} z + y \mathbin{\|\!\lfloor} z$

PA7 $ax \mathbin{\|\!\lfloor} y = a(x \mathbin{\|\!\lfloor} y + y \mathbin{\|\!\lfloor} x)$

PA8 $a \mathbin{\|\!\lfloor} y = ay$

In their paper, process algebra is defined with alternative, sequential and parallel composition, but without communication. This process algebra PA is extended with communication to yield the theory ACP (Algebra of Communicating Processes) [BER84].

Currently, there are a lot of work and applications realized in the three most well-known process algebras CCS, CSP and ACP. CCS is the first with a complete theory. CSP has a least distinguishing equational theory. More than the other two, ACP emphasizes the algebraic aspect: there is an equational theory with a range of semantical models. ACP also has a more general communication scheme: communication is combined with abstraction in CCS while it is combined with restriction in CSP.

There are a number of important developments after the formulation of the basic process algebras CCS, CSP and ACP. Strong bisimulation and weak bisimulation become the central notion of equivalence in process theory. As an alternative to weak bisimulation, branching bisimulation is proposed in [GLA96]. In between bisimulation and trace equivalence there is a whole lattice of other equivalences [GLA01]. For process algebra based on partial order semantics, see [BEST01]. Structural operational semantics (SOS) becomes the dominant way of providing a model for a process algebra. It is standard practice to provide a new operation with SOS rules, even before an equational characterization is attempted. There are many results based on the formats of these rules. An overview can be found in [ACE01].

Petri Nets

Petri nets are devised by Carl Adam Petri as a tool for modeling and analyzing processes [PET62]. A Petri net consists of places and transitions. Places can contain tokens. The structure of a Petri net is fixed. The state of a Petri net is indicated by the distribution of tokens amongst its places. A transition may only fire if it is enabled. This occurs when there is at least one token at each of its input places. The transitions are then, as it were, "loaded": ready to fire. A transition may fire from the moment it is enabled. As it fires, one token is removed from each input place and one token added to each output place. In other words, as it fires a transition consumes tokens from the input place and produces tokens for the output place.

Petri nets enable processes to be described graphically. They also have a strong mathematical basis and they are entirely formalized. Therefore, it is often possible to make strong statements about the properties of the process being modeled. There are also several analysis techniques and tools available which can be applied to analyse a given Petri net. Currently, the model proposed by Carl Adam Petri is expanded upon in many different ways. The three most important types of extension are: the color extension, the time extension and the hierarchical extension. With these extensions, it is possible to model complex processes in an accessible way.

2.4.2 Message Sequence Charts

Message Sequence Charts (MSCs) [MSC96] is a graphical and textual language for description and specification of interaction between entities of a communicating system. MSCs can be used for requirement specification, interface specification, simulation and validation, test case specification and documentation. The core language of Message Sequence Charts is called Basic Message Sequence Charts. A Basic Message Sequence Chart is a finite collection of instances. An instance is an abstract entity on which message outputs, message inputs and local actions may be specified. An instance is denoted by a vertical axis. The time along each axis is running from top to bottom. The events specified on an instance are totally ordered in time; no notion of global time is assumed. No two events on an instance are executed at the same time. An instance is labeled with a name, the instance name. This name is placed above the axis representing the instance. A local action is denoted by a box on the axis with the action text placed in it. A message between two instances is represented by an arrow which starts at the sending instance and ends at the receiving instance. A message is split into a message output and a message input. A message sent by an instance to the environment is represented by an arrow from the sending instance to the exterior of the Message Sequence Chart. A message received from the environment is represented by an arrow from the exterior of the Message Sequence Chart to the receiving instance. A message may be labeled with a parameter list. The parameter list is denoted between brackets after the message name.

The language is standardized in the standard ITU-T Z.120 of the International Telecommunication Union [MSC96]. The standard describes an abstract syntax, a graphical syntax and a textual syntax of the language. The semantics of MSCs is given via a translation originally developed by S. Mauw and M. A. Reniers in [MAUW94]. It transforms the textual representation of MSCs into a process algebra. Section 2.5 introduces their approach in detail.

There are also other ways to define formal semantics for MSCs. J. Grabowski et al. define a formal MSC semantics based on Petri nets [GRA93]. They introduce a special class of Petri nets, the so-called labeled occurrence nets. The translation of MSCs into labeled occurrence nets is defined. Based on this, they provide a semantics for MSC composition and decomposition. Stefan Heymer [HEY00] develops a non-interleaving semantics based on Petri net components for MSC'96 [MSC96] and MSC-2000 [MSC00].

Bengt Jonsson et al. present an execution semantics for a significant of the MSC-2000 standard [JON01]. The semantics has the form of an Abstract Execution Machine, which can either accept or to generate sequences of events that are consistent with a given MSC. In comparison with Petri net semantics, it gives a more direct semantics without translation to an intermediate formalism.

Peter B. Ladkin et al. propose an approach based on finite automata [LAD92] [LAD95]. Formally, a single MSC can be interpreted as a graph with two sorts of edges. The nodes represent communication events, e.g. message sending and message consumption. This graph can be interpreted as a global state transition graph, containing all possible global states specified by the MSC. It corresponds to an automaton without explicitly defined end states.

R. Alur et al. define a semantics for MSC based on partial order methods [ALUR96]. They use semantics definitions that correspond to the visual order of events.

2.4.3 Statecharts

Statecharts is a visual language for specifying the behavior of reactive system [HA87]. The language extends the notation of finite-state machines with concepts of (i) *hierarchy*, so that one may speak of a state as having sub-states, (ii) *concurrency*, thereby allowing the definition of systems having simultaneously active subsystems, and (iii) *priority*, so that one may express that certain system activities have precedence over others. Statecharts is popular among engineers as a design notation for embedded systems.

The survey [BEEK94] lists twenty different statecharts semantics. The first formal semantics appeared in [HAR87]. A. Pnueli et al. [PNU91] and C. Huizing [HUI91] [HUIZ91] discuss which properties statecharts should have and how to design a semantics to obtain them. The first compositional semantics is given in [HOO92].

G. Lüttgen et al. present a process algebraic approach to define a compositional semantics for Statecharts [LUT99]. They define a simple process algebra called Statecharts Process Language (SPL), which is expressive enough for encoding Statecharts in a structure-preserving and semantics-preserving manner. They also establish that the behavioral equivalence bisimulation, when applied to SPL, preserves Statecharts semantics.

A.C. Uselton et al. also focus on achieving a compositional semantics for Statecharts by referring to process algebras [USE94] [USEL94]. They provide a translation of statecharts into a process algebra with state refinement [USE93], a new operator for capturing hierarchy in concurrent systems. The semantics of a statechart is then given by the Labeled Transition System (LTS) of its translation, as defined by the process algebra's structural operational semantics (SOS). The faithfulness of the translation is demonstrated showing that the respective LTSs of a statechart automaton $\mathcal{A}$ and its corresponding process algebra term $t_{\mathcal{A}}$ are isomorphic. R. Eshuis et al. [ESH00] propose a formal real-time semantics for UML statecharts aimed at the requirements level. This semantics is based on the Labeled Transition System (LTS). It is an adaptation of the statemate statechart semantics, with local variables, real time, identifier addressing, point-to-point communication, synchronous communication and dynamic object creation and deletion.

E. Mikk et al. [MIKK97] give a formal semantics definition for statecharts as implemented in statemate and described in [HAR96]. They use the Z notation [SPI92] rather than "standard mathematics" to formalize statecharts semantics. This allows to structure the definition of the formal semantics and to use tools like type-checkers.

2.4.4 Activity Diagrams

UML activity diagrams are special cases of UML state diagrams, which in turn are graphical representations of state machines. The state machine formalism as defined in the UML is a variant of Harel's statecharts [HAR96].

The issue of defining a precise semantics of UML is the subject of intensive investigations. Unfortunately, within this stream of research, activity diagrams have received relatively little attention. There are some efforts attempting to fill this gap. E. Börger et al. define an algebra semantics of the core constructs of activity diagrams [BOR00]. However, they do not deal with features such as synchronization states, dynamic invocation and deferred events. In this regard, the formalization given in [ESH01] is more complete. Based on the statemate semantics of statecharts [HAR96], this formalization covers all activity diagrams constructs (except synchronization states and swimlanes), and considers issues such as data manipulation. The authors however do not formalize syntactical constraints such as the well-formedness rules linking forks with joins, which are essential to avoid some deadlocking situations.

Gehrke et al. [GEH98] give a semantics by mapping activity diagrams into Petri nets. Their semantics does not deal with data or time. It does not model the environment. H. Störrle defines a denotational semantics for activity diagram based on Petri nets [ST03]. This semantics excludes data type annotations and all features based on them, but includes all kinds of control flow, including non well-formed concurrency and, particularly, procedure calling.

2.4.5 Sequence Diagrams

The UML sequence diagram is a variant of Message Sequence Charts. It is extended with constructs for the creation and deletion of objects as well as for synchronous and asynchronous communication. In UML 2.0 [UML20], some features of MSCs are integrated into UML sequence diagrams. UML 2.0 interactions and MSC-2000 [MSC00] are very similar [HAU05].

H. Störrle [STO03] define a straightforward denotational semantics of interactions in the new UML 2.0 standard. The semantics closely follows the new standard in that it defines a partial order semantics for plain interactions, embedded in a trace semantics of CombinedFragments. He also defines a Petri net semantics for the sequence diagram [STO99].

X. Li et al. define a formal semantics of the UML sequence diagram [LI04]. They provide formal semantic models for sequence diagrams directly and then provide the combination of the different models for consistency checking. The static semantics of the sequence diagram is defined as the conjunction of all its messages in the structure tree. The dynamic semantics is defined in terms of the state transitions that are carried out by the method invocations in the diagram. When a message is executed, it must be consistent with system state, i.e., object diagram and the state diagrams of its related objects. The semantics captures the consistency between sequence diagrams with class diagrams and state diagrams.

2.4.6 Use Cases

Ivar Jacobson's work has clearly the greatest influence on defining use cases and establishing the boundaries of what constitutes a use case formalism. He discusses formalizing use case modeling and suggests two directions for further development of use cases [JAC95]. The first direction addresses process and human related concerns. The other addresses model and language related concerns. He provides descriptions for several use case formalisms. The basic use case consists of structured English description which includes alternative and exceptional behavior. Specific scenarios may also be developed for use during development to explain different perspectives of use. Static descriptions for each actor class and each use case class may also be provided. The second formalism is a class association. These include the use case *uses* and *extends* stereotypes and actor inheritance hierarchies. The third type is an interaction diagram, which shows the different paths that the conversation will follow including iteration, repetition, branching, and parallelism. Preconditions and postconditions, such as that the user is logged on to the system, may also be specified. The fourth type is a contract, which specifies an objects interface in detail. Actors may provide a contract that involves multiple use cases. Conversely, a use case may provide a contract that multiple actors use. The fifth formalism type is a state-transition diagram in which a stimulus from an actor will cause the use case to leave its current state and perform a transaction. Each transaction is associated with preconditions and postconditions.

The survey [HUR97] lists thirty-one different approaches for describing and formalizing use cases and twenty-six related works that have an indirect bearing on use case formalisms. It also provides a comparison of the approaches with respect to their focus and representation formats.

In spite of the inclusion of many formalisms into the UML specification, detailed mappings of use cases to other modeling constructs that implement these use cases, and the elaboration of use cases for requirements gathering and domain analysis is ambiguous. There is no formally defined use case semantics.

Most of the recent work in the use case field pertains to use cases in UML. K. G. van den Berg et al. present a control-flow semantics of use cases using flowgraphs [BERG99]. The control-flow for five kinds of use cases is analysed: for common use cases, variant use cases, component use cases, specialized use cases and for ordered use cases. Perdita Stevens uses a Labeled Transition System (LTS) to formally interpret use cases as sequences of actions in [STV01]. H. Ledang et al. propose an approach to translate use case to B specifications [LED02]. In [BUT97], a specification of use cases in the Z notaion [SPI92] is given. The focus is on understanding use cases, not on instrumenting them for the specification in combination with the Z notation. Grieskamp et al. formalize use cases using the Z notation [GRI00]. They extend this work and specify use cases in Abstract State Machine Language [GRI01]. Dranidis et al. [DRA03] propose the specification of use cases with X-machines [EIL74]. They present a method for transforming use case text into its corresponding X-machine model. Hassine et al. define an operational semantics for use case maps [HAS05].

2.5 A Process Algebra for Basic Message Sequence Charts

As mentioned in Section 2.4.2, Message Sequence Charts are standardized. The semantics of MSCs is given via a translation originally developed by S. Mauw and M. A. Reniers in [MAUW94]. This section introduces process algebras that they use to specify Basic Message Sequence Charts. This helps understand the use case semantics defined in Section 3.2, which is based on these process algebras.

2.5.1 Process Algebra PA$_\varepsilon$

Process algebra PA_ε is an algebraic theory describing process behavior [BAE90] [BER84]. It is given by a signature $\sum_{PA_\varepsilon}$ defining the processes and a set of equations E_{PA_ε} defining the equality relation on these processes. This section gives the signature $\sum_{PA_\varepsilon}$ and the set of equations E_{PA_ε}. It describes the semantics of Basic Message Sequence Charts using an example.

The Signature of PA_ε

A signature $\sum$ is a set of constant and function symbols. For every function symbol in the signature its arity is specified. The terms associated to a signature $\sum$ and a set of variables V are defined as follows.

Let $\sum$ be a signature and let V be a set of variables. Terms over signature $\sum$ with variables form V are defined inductively by:

1. $v \in V$
2. if $c \in \sum$ is a constant symbol, then c is a term.
3. if $f \in \sum$ is an n-ary (n $\geq$ 1) function symbol and $t_1, \cdots, t_n$ are terms, then $f(t_1, \cdots, t_n)$ is a term.

The signature E_{PA_ε} consists of:

1. the special constants δ and ε
2. the set of unspecified constants A
3. the unary operator $\surd$
4. the binary operators $+$, $\cdot$, $\|$, $\lfloor\!\lfloor$

The special constants δ denotes the process that has stopped executing actions and cannot proceed. This constant is called *deadlock*. The special constant ε denotes the process that is only capable of terminating successfully. It is called the *empty process*.

The elements of the set of unspecified constants A are called *atomic actions*. These are the smallest processes in the description. This set is considered a parameter of the theory.

The binary operators $+$ and $\cdot$ are called the *alternative* and *sequential* composition. The *alternative* composition of the processes x and y is the process that either executes process x or y but not both. The *sequential* composition of the processes x and y is the process that first executes process x, and upon completion thereof starts with the execution of process y.

The binary operator $\|$ is called the *free merge*. The *free merge* of the processes x and y is the process that executes the processes x and y in parallel. For a finite set $D = \{d_1, \cdots, d_n\}$, the notation $\|_{d\in D}\mathrm{P(d)}$ is an abbreviation for $\mathrm{P(d_1)} \| \cdots \| \mathrm{P(d}_n)$. If $D = \emptyset$ then $\|_{d\in D}\mathrm{P(d)} = \varepsilon$. For the definition of the merge, two auxiliary operators are used. The *termination operator* $\surd$ applied to a process x signals whether or not the process x has an option to terminate immediately. The binary operator $\lfloor\!\lfloor$ is called the *left merge*. The *left merge* of the processes x and y is the process that first has to execute an atomic action from process x, and upon completion thereof executes the remainder of process x and process y in parallel.

The Equations of PA_ε

The set of equations E_{PA_ε} of PA_ε specifies which processes are considered equal. An equation is of the form $t_1 = t_2$, where $t_1, t_2 \in T(E_{PA_\varepsilon}, V)$. For $a \in A \cup \{\delta\}$ and $x, y, z \in V$, the equations of PA_ε are given as follows:

A1: $x + y = y + x$
A2: $(x + y) + z = x + (y + z)$
A3: $x + x = x$
A4: $(x + y) \cdot z = x \cdot z + y \cdot z$
A5: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
A6: $x + \delta = x$
A7: $\delta \cdot x = \delta$
A8: $x \cdot \varepsilon = x$

A9: $\varepsilon \cdot x = x$

TM1: $x \parallel y = x \mathbin{\underline{\|}} y + y \mathbin{\underline{\|}} x + \surd(x) \cdot \surd(y)$
TM2: $\varepsilon \mathbin{\underline{\|}} x = \delta$
TM3: $a \cdot x \mathbin{\underline{\|}} y = a \cdot (x \parallel y)$
TM4: $(x + y) \mathbin{\underline{\|}} z = x \mathbin{\underline{\|}} z + y \mathbin{\underline{\|}} z$

TE1: $\surd(\varepsilon) = \varepsilon$
TE2: $\surd(a \cdot x) = \delta$
TE3: $\surd(x + y) = \surd(x) + \surd(y)$

Axioms A1-A9 are well known. The axioms TE1-TE3 express that a process x has an option to terminate immediately if $\surd(x)=\varepsilon$, and that $\surd(x)=\delta$ otherwise. In itself the termination operator is not very interesting, but in defining the free merge this operator is needed to express the case in which both processes x and y are incapable of executing an atomic action. Axiom TM1 expresses that the free merge of the two processes x and y is their interleaving. This is expressed in the three summands. The first two state that x and y may start executing. The third summand expresses that if both x and y have an option to terminate, their *merge* has this option too.

2.5.2 Process Algebra PA$_{BMSC}$

The process algebra PA$_{BMSC}$ is defined in [MAUW94]. It extends PA$_\varepsilon$ by specifying the set of atomic actions and by introducing the auxiliary operator λ_M.

Atomic Actions

There are different atomic actions in Basic Message Sequence Charts, which are listed below:

- The execution of an action aid by instance i: action(i, *aid*)
- The sending of a message m by instance s to instance r : *out*(s, r, m)
- The sending of a message m by instance s to the environment: *out*(s, *env*, m)
- The receiving of a message m by instance r from instance s : *in*(s, r, m)
- The receiving of a message m by instance r from the environment: *in*(*env*, r, m)

State Operator λ_M

A Basic Message Sequence Chart specifies a finite set of instances communicating by sending and receiving messages. A message is divided into an output and an input. The correspondence between message outputs and message inputs is defined uniquely by message name identification.

The operator λ_M is introduced to enable only those execution paths respecting the following constraint: a message output cannot be executed before the corresponding message input has been executed. The operator λ_M is an instance of the state operator in [BAE90].

2.5.3 Semantics

The general idea is that the semantics of a Basic Message Sequence Chart is the free merge of the semantics of its constituent instances. The state operator λ_M is used to rule out all interleavings where a message output is preceded by the corresponding message input. For example, Figure 2 shows an example of a Basic Message Sequence Chart. It consists of two instances: a_1 and a_2, which exchange two messages: i and j.

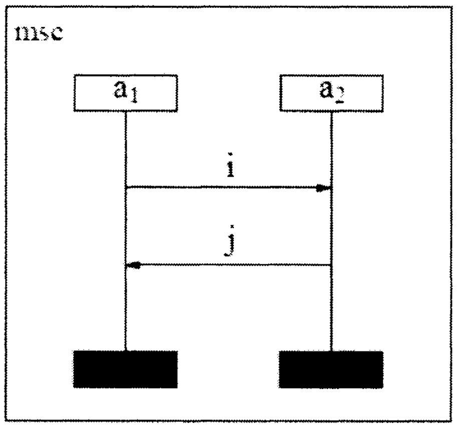


Figure 2: Example: Basic Message Sequence Chart

The semantics of instance a_1 is $out(a_1, a_2, i) \cdot in(a_2, a_1, j)$. The semantics of instance a_2 is $in(a_1, a_2, i) \cdot out(a_2, a_1, j)$. The semantics of this Basic Message Sequence Chart is as follows.

$$\lambda_M(out(a_1, a_2, i) \cdot in(a_2, a_1, j) \parallel in(a_1, a_2, i) \cdot out(a_2, a_1, j))$$

After some calculations, it appears some unwanted execution traces where the message output comes before its message input. The λ_M operator is applied to remove all these unwanted traces. As a result, the semantics of this Basic Message Sequence Chart equals to:

$$out(a_1, a_2, i) \cdot in(a_1, a_2, i) \cdot out(a_2, a_1, j) \cdot in(a_2, a_1, j)$$

Chapter 3

Refactoring Use Case Models

This chapter describes the core work of refactoring use case models. Section 3.1 defines a use case metamodel, which formalizes use cases and defines related terminologies. Section 3.2 defines a process algebra semantics for the use case model. This semantics makes it possible to compare the behavior of the use case model in a formal manner. Section 3.3 introduces eight invariants required to preserve the behavior of the use case model. This is essential for verifying the behavior preserving property of use case refactorings. Section 3.4 presents the list of use case refactorings briefly. It presents a template for defining each use case refactoring.

3.1 Use Case Metamodel

Different people are using use cases differently. Use cases are the medium to model user requirements for requirements engineering. They are used to guide the design of communicating objects to satisfy functional requirements for software engineers. Although UML attempts to formalize use cases, there is still a lot to argue about the use and semantics of use cases.

This section defines a use case metamodel, which represents a new perspective on the use case formalization. This metamodel is a basis for refactoring use case models within the thesis because it provides the specification of use case terminologies.

The rest of this section is organized as follows. Section 3.1.1 provides an overview on the use case metamodel. It gives the definition for elements at different levels: environment level, structure level and event level. Section 3.1.2 defines relationships at the environmental level, which include the use case relationship, actor relationship, task relationship and goal relationship. Section 3.1.3 defines the episode model, which specifies the behavior of the use case at the structure level. Section 3.1.4 defines the event model, which specifies the episode behavior. Section 3.1.5 discusses this use case metamodel.

3.1.1 Overview

Figure 3 shows an overview of the use case metamodel defined in this research. It is intended to provide support to requirements engineering and software engineering. This metamodel incorporates Regnell's use case specification [REG99] as well as other people's work in use case modeling. The use case model mainly consists of actors, use cases and related episodes and events. This metamodel introduces the concept of goal to support the goal-oriented requirements engineering and the concept of task to support the task-oriented requirements engineering.

This metamodel can be viewed from different levels. At the environment level, the use case is related to entities external to the intended system. At the structure level, the internal structure of a use case is revealed together with its different variants and parts. The event level represents a lower abstraction level where individual events are characterized. Elements within this metamodel for each level are defined as follows.

- **Environment Level**: The key elements are *use case*, *service*, *user*, *actor*, *goal* and *task*.

  A *use case* is a system usage scenario characteristic of a specific actor. A use case models a usage situation where one or more *services* of the target system are used by one or more *users* with the aim to accomplish one or more *goals*. A *use case* may either model a successful or an unsuccessful accomplishment of *goals*. A *use case* is used to define the behavior of a system without revealing the entity's internal structure. Each *use case* specifies a sequence of activities that the entity can perform, interact with *actors* of the entity.

  A *use case* can be concrete or abstract. A *concrete use case* is a use case that can be instantiated. An *abstract use case* is a use case that cannot be instantiated on its own. It is only meaningful to describe the behavior which is common to other use cases.

  An *actor* is a specific role played by a system user, and represents a category of users that demonstrate similar behavior when using the system. It may be considered as playing a separate role with regard to each *use case* with which it communicates.

  An *actor* can be concrete or abstract. A *concrete actor* is an actor that can be instantiated. An *abstract actor* is an actor that cannot be instantiated. It represents a role that no user can play directly. It must be inherited by other concrete actors. An abstract actor is used to capture the common characteristics between those concrete actors.

  A *user* is regarded as an instance of an actor. The user can be human beings, and other external systems or devices communicating with the system. One user may appear as several instances of different actors depending on the context.

  A *service* is a package of functional entities (features) offered to the users in order to satisfy one or more goals that the users have.

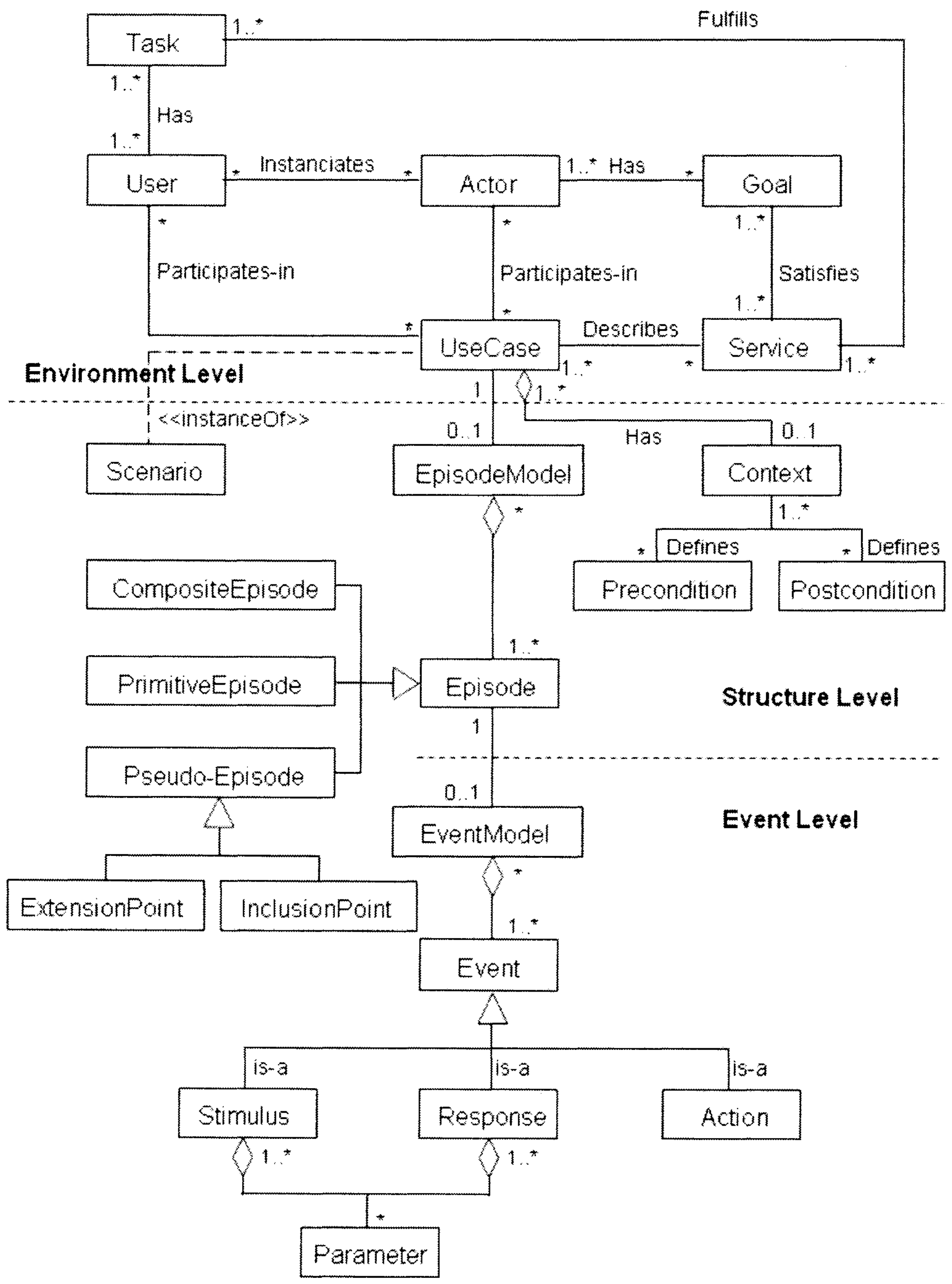


Figure 3: Use Case Metamodel

Goals are objectives that users have when using services of a target system. Goals are often used to categorize users into actors.

Task describes what the user will do with the software system.

- **Structure Level**: The key elements are *scenario*, *episode* and *context.*

  A *scenario* is a specific and bound realization of a use case described as a sequence of a finite number of events with linear time order.

  A use case may be divided into coherent parts, called *episodes*. These coherent episodes reveal the internal structure of a use case, and describe the functionalities of a use case. The same episode can occur in different use cases.

  An *episode* can be concrete or abstract. A *concrete episode* is a coherent part in the *concrete use case*. An *abstract episode* represents the common behavior among different episodes. It does not occur in any use case directly. There can be a *generalization* relationship between an abstract episode and a concrete episode. The *generalization* relationship implies that the child episode contains all the attributes, sequences of behavior defined in the parent episode. The child episode may also add new behavior sequences.

  A use case may have a *context*, which demarcates the scope of the use case and defines its *preconditions* and *postconditions*. A *precondition* is the property of the environment and the target system that needs to be fulfilled in order to invoke the use case. A *precondition* describes the state or status of the system before a use case executes. A *postcondition* is the property of the environment and the target system at use case termination. A *postcondition* describes the status of the system as a result of the use case completing.

- **Event Level**: The key element is the *event.*

  An *event* is the specification of a significant occurrence that has a location in time and space. The difference between an event and an episode is that an event takes no duration whereas an episode does.

  There are three types of events: *stimulus*, *response* and *action*. A *stimulus* is the message from users to the target system. A *response* is the message from the target system to users. An *action* is the target system intrinsic event which is atomic in the sense that there is no communication between the target system and users that participate in the use case. The *stimulus* and *response* can take *parameters*, which carry data to and from the target system.

3.1.2 Environmental Entity Relationships

This section describes relationships at the environmental level, which includes the use case relationship, actor relationship, task relationship and goal relationship. The meaning of each relationship is defined in detail.

Use Case Relationship

The metamodel for the use case relationship is shown in Figure 4. There are six use case relationships: *inclusion*, *extension*, *generalization*, *similarity*, *equivalence*, and *precedence*.

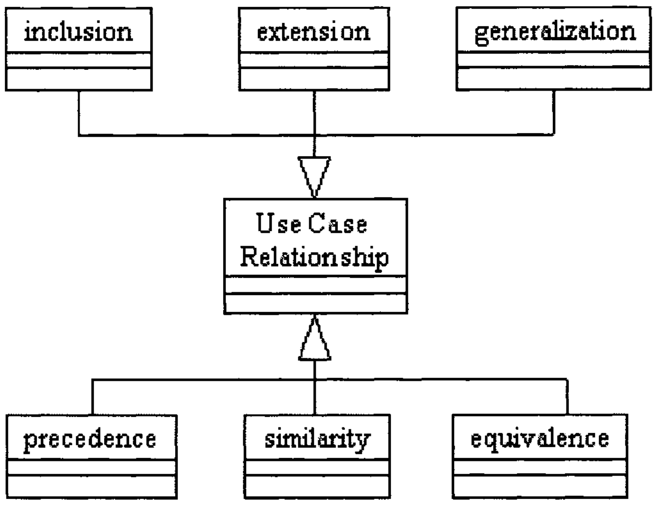


Figure 4: Metamodel for the Use Case Relationship

An *inclusion* relationship between two use cases means that the behavior defined in the target use case is included at one location in the sequence of behavior performed by an instance of the base use case. It specifies that one use case explicitly incorporates the behavior of another at the given point. This given point is called the inclusion point. When one use case instance reaches the inclusion point, it performs all the behavior described by the included use case and then continues according to its original use case. One use case may be included in several other use cases and one use case may include several other use cases. The included use case may not be dependent on the base use case. In that sense the included use case represents encapsulated behavior which may easily be reused in several use cases. Moreover, the base use case may only be dependent on the results of performing the included behavior and not on the structure, like attributes and associations, of the included use case.

An *extension* relationship between two use cases specifies that one use case extends the behavior of another at the given extension point. One use case extends another by introducing alternative or exceptional processes. It defines that instances of a use case may be augmented with some additional behavior defined in an extending use case. The extension relationship contains a condition and references a sequence of extension points in the target use case.

A *generalization* relationship between two use cases implies that the child use case contains all the attributes, sequences of behavior, and inclusion/extension points defined in the parent use case, and participates in all relationships of the parent use case. The child use case may also add new behavior sequences.

A *similarity* relationship between two use cases defines that one use case corresponds to or is similar to or resembles another in some unspecified ways. Similarity is a relationship often noted early in use case modeling. It provides a way to carry forward insight about relationships among use case even when the exact nature of the relationship is not yet clear.

An *equivalence* relationship between two use cases defines that one use case is equivalent to another, that is, serves as an alias. Equivalence flags those cases where a single definition can cover what are, from the user's perspective, two or more different intensions. It makes it easier to validate the model with users and customers while also assuring that only one design will be developed.

A *precedence* relationship between two use cases defines that one use case is sequenced (appended) to the behavior of the preceding use case.

Actor Relationship

The metamodel for the actor relationship is shown in Figure 5. It defines the *generalization* relationship between actors. Two or more actors may have commonalities, i.e. communicate with the same set of use cases in the same way. The commonality is expressed with *generalizations* to another (possibly abstract) actor, which models the common role(s). This means that the child actor will be able to play the same roles as the parent actor, i.e. communicate with the same set of use cases, as the parent actor. If the parent actor interacts with the use case directly, both the parent actor and the child actor are concrete actors.

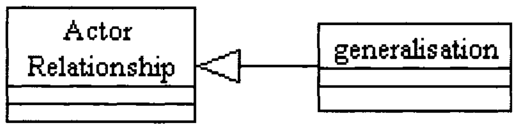


Figure 5: Metamodel for the Actor Relationship

Task Relationship

The metamodel for the task relationship is shown in Figure 6. There are two task relationships: *inclusion* and *generalization*.

An *inclusion* relationship between two tasks defines that one task contains another task, which is a subtask. One subtask may be included in several other tasks and one task may include several

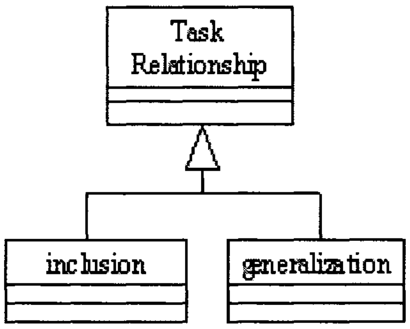


Figure 6: Metamodel for the Task Relationship

other subtasks. Two or more tasks may have commonalities, i.e. contain the same set of subtasks. This commonality is expressed with the *generalization* to another task, which models the common task(s).

Goal Relationship

The metamodel for the goal relationship is shown in Figure 7. Similar to task relationships, there are two goal relationships: *inclusion* and *generalization.*

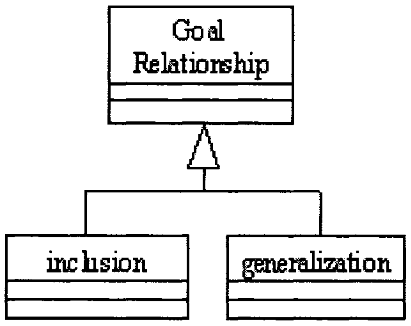


Figure 7: Metamodel for the Goal Relationship

An *inclusion* relationship between two goals defines that one goal may contain another goal, which is a subgoal. One subgoal may be included in several other goals and one goal may include several subgoals. Two or more goals may have commonalities, i.e. contain the same set of subgoals. This commonality is expressed with the *generalization* to another goal, which models the common goal(s).

3.1.3 Episode Model

Episode Metamodel

The behavior of a use case is described in an episode model. Since an *abstract episode* does not occur in any use case directly, only a *concrete episode* can participate in the episode model directly. The episode metamodel is defined as shown in Figure 8.

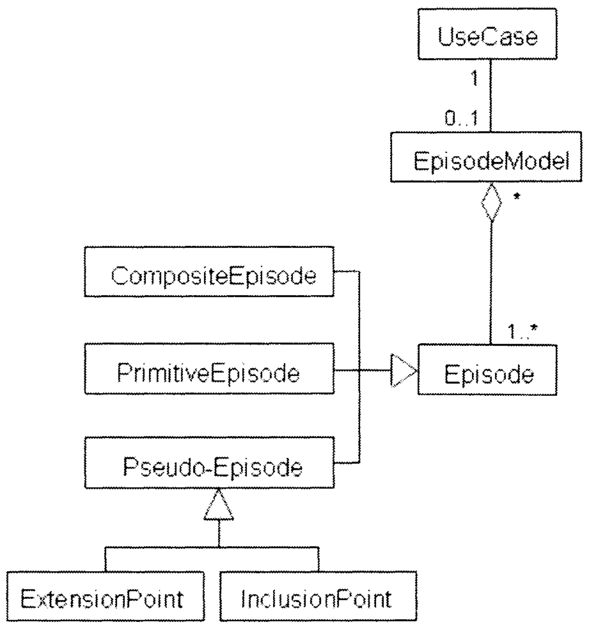


Figure 8: Episode Metamodel

Concrete episodes can be divided into different categories. An episode can be a primitive, composite, or pseudo-episode. An episode that defines the basic activity for implementing the behavior of a use case is considered as a primitive episode. An episode with operators to define a complex activity is considered as a composite episode. There are different operators: *sequence*, *alternation*, *iteration* and *parallel*. One composite episode can relate to other composite episodes or primitive episodes through these operators. The concept of *pseudo-episode* is introduced in this research to indicate an insertion point for the inclusion relationship or an extension point for the extension relationship between use cases.

These episodes are described further as follows:

- **Primitive Episode**: a basic episode. It is the fundamental element in an episode model.
- **Sequence Episode**: a composite episode. Its child episodes are executed one-by-one in the time order. A *sequence* episode is used to describe sequential activities in a use case.

- **Alternation Episode**: a composite episode. Depending on the condition, only one of its child episodes is selected to be executed. The condition is defined in the *alternation* episode. An *alternation* episode is used to describe the alternative course in a use case.

- **Iteration Episode**: a composite episode. Its child episodes are executed multiple times according to the specified parameter. An *iteration* episode is used to describe repetitive activities in a use case. The number of repetitions is specified by giving a number or an interval.

- **Parallel Episode**: a composite episode. There is no time order constraint for its child episodes. In other words, these episodes can be executed in any order. A *parallel* episode is used to describe the parallel activities in a use case.

- **Pseudo-Episode**: a special episode. It is used to indicate the inclusion point or extension point in the base use case when an inclusion or extension relationship exists between the base use case and the related use case.

In order to represent the inherited episode, the thesis introduces the concept of *episode placeholder*. An *episode placeholder* is the alias of an inherited episode. It participates in the episode model, but it depends on the generalization relationship with the parent use case. If the generalization relationship is removed, the episode placeholder is removed correspondingly.

Episode Tree

In this research, an *episode tree* is defined as a graphical representation of the episode model using a tree structure. A parent node in the tree represents a composite episode. A leaf node in the tree represents a primitive episode. The type of a parent node can be *sequence*, *alternation*, *iteration*, or *parallel*, which implies the episode relationship among its child nodes.

By adopting the tree structure, it is very convenient to represent the relationship between a composite episode and related primitive episodes. It is especially convenient to build the tree structure in Java when the use case modeling and refactoring tool is developed. As an example, the episode tree of use case "Withdraw" in the ATM (Automatic Teller Machine) system is shown as follows.

Episode tree for use case "Withdraw"

E0: (sequence)
  E1: Insert and validate the bank card
  E2: Enter and validate PIN
  E3: Select the bank account
  E4: Enter the amount
  E5: (alternation) ATM processes the withdrawal request
    E5.1: (sequence) bank permits the operation
        E5.1.1: Customer gets cash

E5.1.2: ATM updates the account balance
E5.2: Bank denies the operation
E6: ATM ejects the bank card

As shown above, E0 is a *sequence* composite episode. It has six child episodes: E1, E2, E3, E4, E5, and E6. These episodes are executed sequentially. E5 is an *alternation* episode. Either E5.1 or E5.2 is executed. E5.1 is a *sequence* composite episode. The episode composition and decomposition within the episode tree not only make a use case more understandable, but also improve the reusability of the use case model. Moreover, an episode model can be easily translated into an indented textual notation as shown above by a depth-first traversal of the tree.

Well-formedness Rules

In this research, the UML Object Constraint Language (OCL) [UML20] is used to define constraints in the episode metamodel shown in Figure 8. Well-formedness rules for this metamodel are described as follows.

Context UseCase inv A use case can be refined using an episode model. A concrete use case always has an episode model. An abstract use case does not always have an episode model.

Self.EpisodeModel->size() = 0 or
Self.EpisodeModel->size() = 1

Context EpisodeModel inv An episode model can be represented as a tree. Only one composite episode serves as the root of the episode tree.

self.Episode.CompositeEpisode.allInstances-> isUnique(e | e.isRoot = true)

An episode model refines only one use case.
self.UseCase->size() = 1

Context Episode inv An episode is specified as a primitive episode, composite episode, or pseudo-episode in an episode model.

self.EpisodeModel.oclType.elements
(forAll(m | m.stereotype = PrimitiveEpisode
or m.stereotype = CompositeEpisode
or m.stereotype = Pseudo-Episode))

Context CompositeEpisode inv There is only one episode tree for an episode model. The root is a composite episode.

self.allInstances->isUnique(e | e.isRoot = true)

A composite episode is a root node in an episode tree if and only if its parent is empty.
self.parent->isEmpty implies self.isRoot = true

There are four kinds of operators in a composite episode: sequence, alternation, iteration and parallel.
self.operator.type: enum { sequence, alternation, iteration, parallel }

A composite episode consists of one or more composite episodes, primitive episodes, or pseudo-episodes.
let c = self.child
c(forAll (ce | ce.typeOfEpisode = CompositeEpisode
or ce.typeOfEpisode = PrimitiveEpisode
or ce.typeOfEpisode = Pseudo-Episode)

Context PrimitiveEpisode inv The name of PrimitiveEpisode cannot be empty.
not self.name = ‘’

The name of PrimitiveEpisode must be unique.
self.allInstance->forAll(p1, p2 | p1.name = p2.name implies p1 = p2)

Context Pseudo-Episode inv The name of Pseudo-Episode cannot be empty.
not self.name = ‘’

The name of Pseudo-Episode must be unique.
self.allInstance->forAll(p1, p2 | p1.name = p2.name implies p1 = p2)

3.1.4 Event Model

The behavior of an episode is described in an event model. An event metamodel is defined as shown in Figure 9. Similar to the episode model, there are also different operators: *sequence*, *alternation*, *iteration* and *parallel*, which are described further as follows:

- **Sequence Operator**: It is used to describe the sequential event trace. It specifies that events are executed one-by-one in the time order.

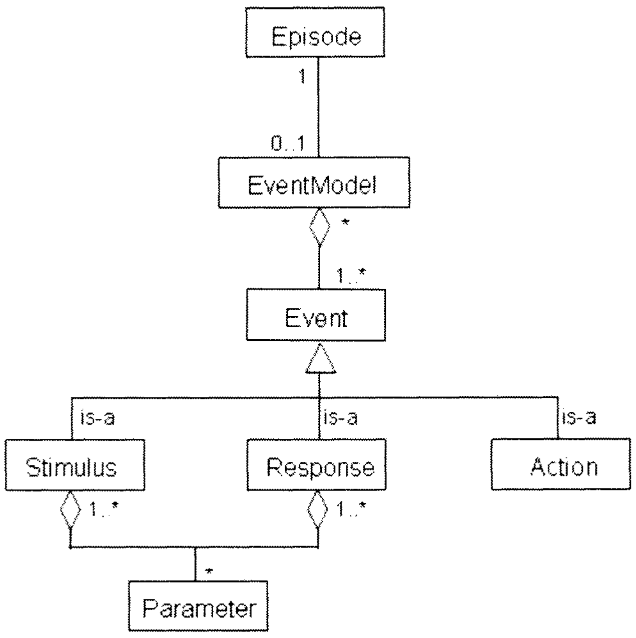


Figure 9: Event Metamodel

- **Alternation Operator**: It is used when a choice can be made between groups of events.
- **Iteration Operator**: It is used when a sequence of events should be repeated. The number of repetitions is specified by giving a number or an interval.
- **Parallel Operator**: It is used when events are executed synchronously.

Based on these operators, an event tree can be constructed as a graphical representation of the event model. It uses a tree structure, which is similar to how the episode tree is created. The difference is that the episode tree uses composite episodes to relate different episodes while the event tree uses the above operators to relate different events. As an alternative, a basic MSC can also be used to represent the event model graphically.

In order to represent the inherited event, the thesis introduces the concept of *event placeholder*. An *event placeholder* is the alias of an inherited event. It participates in the event model, but it depends on the generalization relationship with the parent episode. If the generalization relationship is removed, the event placeholder is removed correspondingly.

3.1.5 Discussion

As the first step in refactoring use case models, use cases have to be formalized. This use case metamodel defines the use case specification using a three-level structure. It defines terminologies used in the use case model. While it is controversial to argue how use cases should be formalized, this metamodel is defined to address different needs in requirements engineering and software engineering.

This metamodel is based on Regnell's use case model [REG99], which is a basis for requirements engineering. This model utilizes three levels for use case modeling. It allows a hierarchical structure and enables graphical representation at different abstraction levels. Different levels are related to each other. This makes it possible to integrate the two disciplines of use case modeling and statistical usage testing to form a seamless transition from requirements models to test models for reliability certification. The event level describes detailed interaction between the system and actors, which is essential for design, implementation and testing. This satisfies needs for software engineering.

Based on Regnell's model, the structure level is refined by introducing the episode model and episode tree within this research. This helps manage and present the use case internal structure effectively. Based on the episode model, the use case model can be restructured by manipulating episodes among different use cases.

The concept of task is introduced in this use case metamodel to support the task-oriented requirements engineering. Task-oriented approaches are typically used in user interface design. The process of gathering and understanding data about the tasks is called task analysis. The first publication about task analysis was probably "Hierarchical Task Analysis (HTA)" [ANN67]. In HTA, tasks are defined as activities that people do to reach a goal. HTA is the foundation for many other methods. Task analysis has been developed gradually since then. Task models are the "product" of the entire task analysis activity. A hierarchical task model is created by breaking the task into increasingly detailed task elements. In this metalmodel, two relationships between tasks are defined, which are "inclusion" and "generalization".

Tasks and goals are related to each other. In this metamodel, the concept of goal is introduced to support the goal-driven requirements engineering [DAR93] [ANT97] [COC97]. Similar to task relationships, two relationships between goals are defined, which are "inclusion" and "generalization".

In this metamodel, a comprehensive set of use case relationships are defined. As shown in Table 2, this metamodel covers use case relationships as described for Objectory [JAC92], SOMA [SEL97], OML [FIR98] and UML (v2.0) [UML20]. Among these relationships, the "uses" relationship is semantically equivalent to the "includes" relationship. The "usage", "composition" and "invokes" relationships can be expressed by the "includes" relationship. Since both "specialization" and "generalization" relationships aim at a hierarchical structure with inheritance, only one of them is kept within this metamodel. Besides these relationships, the "similarity" relationship is added because it provides a way to carry forward insight about relationships among use cases when the exact nature of the relationship is not yet clear. The "equivalence" relationship is added because it can be very useful when a single definition covers two or more different intentions from the user's perspective.

Table 2: Comparison of Use Case Relationships

	Objectory Jacobson	**SOMA Graham**	**OML Firesmith**	**UML Rational**
Use Case Relationship	uses extends	usage composition specialization	invokes precedes	includes extends generalization

3.2 A Process Algebra Semantics

Use cases are used differently by different people. There is no agreed use case semantics. In this research, we need to define a use case semantics to verify behavior preservation of use case refactorings. The thesis presents an execution semantics of use case. Use case semantics is defined as the execution of the use case. The semantics of the use case model is defined as the set of possible executions of all use cases within the model. In essence, the behavior of the use case model is the set of event traces. It is difficult to define use case semantics simply based on the environment level. Detailed sequences of events in the use case are only specified in its structure level and event level.

Process algebra PA_{BMSC} [MAUW94] can be used to define the episode semantics. Section 2.5 shows how the semantics of a Basic Message Sequence Chart can be defined by process algebra PA_{BMSC}. An episode can be represented by a Basic Message Sequence Chart. Andersson et al. illustrate how to formalize use cases through Message Sequence Charts [AND97]. They define the use case model in a layered manner, which is the basis of the three-level structure used in the use case metamodel within the thesis. Different layers provide different information and have different degrees of details. There are three levels in their model: system level, structure level and basic level. Each level consists of a set of diagrams describing the functionality of the system. The system level expresses the functional view of the system. It describes the full collection of actors, use cases and reuse. The structure level describes the use case behavior without going into details. The basic level shows the detailed interaction between the system and the actors. It corresponds to the event level in the use case metamodel defined in the thesis. The graphical notation of this approach is based on Message Sequence Charts (MSCs). They add some additional syntax to the MSC standard.

Since process algebra PA_{BMSC} is already available for us to use, we choose process algebra to define use case semantics for convenience. We do not need the full power of process algebra. We only use it to specify the set of event traces in the use case model. Concurrency is not concerned in the thesis. Process algebra provides a natural mapping for the sequence of events in the use case model. Relationships between events are mapped into binary operators in process algebra directly. Sequence operator in the event model, for example, is translated into sequential composition operator in process algebra.

Based on the episode semantics defined by process algebra PA_{BMSC}, the semantics of a use case is defined in terms of the process algebra semantics of the episode model. Each use case corresponds to a process algebra term which defines what happens when the use case is executed. The semantics of the use case model is defined as the alternation of the process algebra terms of all use cases which can be executed.

This section introduces the process algebra semantics for the use case model. Section 3.2.1 defines the semantics for the single use case. It analyses different episode relationships. It also discusses the semantics of the incomplete use case. Section 3.2.2 defines the semantics of the use case model. It analyses different use case relationships.

3.2.1 Use Case Semantics

The use case semantics is the interaction between use case and its actors. Detailed interactions between use cases and actors can be specified at the event level. An episode can be represented by a Basic Message Sequence Chart, which illustrates these interactions. The semantics of a Basic Message Sequence Chart can be defined by a process algebra term using PA_{BMSC} [MAUW94]. Since a use case consists of a set of episodes and its behavior is represented by its episode model, the semantics of a single use case can be defined. The following section illustrates how the semantics of a single use case can be specified based on different episode relationships. It also illustrates how to deal with the incomplete use case.

Semantics of a Use Case

Given a concrete use case U, it consists of a finite set of primitive episodes $\{e_1, \cdots, e_n\}$. The episode model of use case U can be defined based on these episodes and their relationships. The episode model can be represented by an episode tree. Let composite episode E be the root node of this episode tree, P_E be the process algebra term of episode E. The semantics S_U of use case U can be represented by P_E, which describes the process of the whole use case U.

Let $\{P_{e_1}, \cdots, P_{e_n}\}$ be a set of process algebra terms corresponding to primitive episodes $\{e_1, \cdots, e_n\}$. Process algebra term P_E can be represented by $\{P_{e_1}, \cdots, P_{e_n}\}$ based on episode relationships. This is analysed below in detail.

(1) $n = 1$: There is only one primitive episode: $\{e_1\}$. Obviously, process algebra term P_E equals to process algebra term P_{e_1}.

(2) $n = 2$: There are only two primitive episodes: $\{e_1, e_2\}$. Depending on the relationship between e_1 and e_2, process algebra term P_E can be defined as follows:

- $P_E = P_{e_1} \cdot P_{e_2}$ if there is a *sequence* relationship between episodes e_1 and e_2, e_1 precedes to e_2.
- $P_E = P_{e_1} + P_{e_2}$ if there is an *alternation* relationship between episodes e_1 and e_2.

- $P_E = (P_{e_1} \cdot P_{e_2})^{i}$, where i is the interval or the number of repetitions, if there is an *iteration* relationship between episodes e_1 and e_2, e_1 precedes to e_2.
- $P_E = P_{e_1} \parallel P_{e_2}$ if there is a *parallel* relationship between episodes e_1 and e_2.

(3) $n > 2$: A composite episode can be used to represent two episodes based on their relationships. If this composing process is applied recursively, the episode model of the use case U can be represented by two composite episodes: $\{E_1, E_2\}$. So process algebra term P_E can be defined by process algebra terms of $\{E_1, E_2\}$ using the approach above ($n = 2$).

For example, based on the episode model presented for the use case "Withdraw" in Section 3.1.3, the semantics of this use case can be represented by:

$$P_{Withdraw} = P_{E_1} \cdot P_{E_2} \cdot P_{E_3} \cdot P_{E_4} \cdot ((P_{E_{5.1.1}} \cdot P_{E_{5.1.2}}) + P_{E_{5.2}}) \cdot P_{E_6}$$

Incomplete Use Case

A use case can be incomplete, which means that the information at the structure level and event level is not be fully defined. This is very common in practice due to various reasons. For example, the requirement is not clear enough, or people do not intend to specify too much details. In order to handle this issue, the whole use case is modeled as a single episode, which is represented by the root node of the episode tree. This is essentially the anonymous entity identified only by name and representing "Any", the top, most general behavior. Then the structure of the use case can be refined stepwise.

As an example, let us revisit the episode model of use case "Withdraw" in Section 3.1.3. The semantics of this use case is defined as follows depending on how much structural information is specified.

- Primitive episodes are not specified: the whole use case is modeled as a single episode E_0. So the semantics of this use case is represented by P_{E_0}.
- All episodes are specified except E_5: the following episodes are defined: E_1, E_2, E_3, E_4, and E_6. The episode E_5 is not fully defined. It represents the most general behavior on processing the withdrawal request. In this stage, the semantics of this use case is defined as:

  $$P_{Withdraw} = P_{E_1} \cdot P_{E_2} \cdot P_{E_3} \cdot P_{E_4} \cdot P_{E_5} \cdot P_{E_6}$$

- Episode E_5 is refined by defining episodes $E_{5.1}$ and $E_{5.2}$. The episode $E_{5.1}$ needs to be refined further, so it represents the general behavior that bank permits the operation. In this stage, the semantics of this use case is defined as:

  $$P_{Withdraw} = P_{E_1} \cdot P_{E_2} \cdot P_{E_3} \cdot P_{E_4} \cdot (P_{E_{5.1}} + P_{E_{5.2}}) \cdot P_{E_6}$$

The above approach allows stepwise refinement of the use case. The episode model must conform to well-formedness rules defined in Section 3.1.3. If the relationship of an episode with other episodes in the use case is unknown, this episode cannot be used to construct the episode model.

Similarly, when the event model of an episode is not defined, the process algebra term of this episode represents its most general behavior. In the above example, P_{E_1} represents the most general behavior of the episode E_1.

3.2.2 Semantics of the Use Case Model

The semantics for a use case model deals primarily with the concepts of actor, use case, and episode as the tool and the refactorings mainly treat these core concepts. The metamodel is cast more broadly to indicate how tasks, goals, and services might be handled in the future. The semantics for the event level is provided from the literature in order to support the future refinement of use cases models at the event level.

A use case model consists of a set of actors, a set of use cases, and the related episodes (and implicitly the related event models). The key semantics is the semantics of a use case defined above in terms of the process algebra semantics of the episode model. Each use case corresponds to a process algebra term which defines what happens when the use case is executed. The semantics of the use case model is defined as the set of possible executions, that is, the alternation of the terms P_u, over all use cases u which can be executed.

Note that we distinguish between abstract use cases and concrete use cases. Abstract use cases can never be instantiated in their own right, so are never executed. They contribute to the definition of the semantics of concrete use cases. Concrete use cases can be instantiated and executed.

So, the semantics of a use case model is the alternation of P_u for all concrete use cases u. Since merging independent use cases u_1 and u_1 gives a use case u with term $P_u = P_{u_1} + P_{u_2}$, we wish to view a use case model as a collection of independent concrete use cases. Hence below we discuss the impact of the use case relationships on defining this set of independent concrete use cases.

For convenience, we define the semantics of an actor to be the semantics of the set of use cases for which it is the primary actor, that is, for which it initiates the execution.

The process algebra term P_u is defined as the term corresponding to the episode model of the use case u. The episode model is a composite episode representing episode tree of operators, subtrees, and leafs. The episode operators translate naturally to operators in the process algebra. The semantics is a term $P_u(e_1, e_2, ..., e_n)$ defined over the primitive episodes in the episode model.

The episode model is augmented by pseudo-episodes and placeholders. The pseudo-episodes represent inclusion points and exclusion points. The placeholders represent the episode which is inherited from the parent use case by a child use case when using the generalization relationship between use cases. The parent episode model is also an episode tree with a corresponding process algebra term. These pseudo-episodes and placeholders can be represented by process algebra variables, which in turn can be substituted for by the appropriate term of the process algebra.

For example, suppose u is a use case with an inclusion point. Then the process algebra term for u involves a free variable I corresponding to the inclusion point, so the term is $P_u(I)$. If the included use case u_1 has process algebra term $t = P_{u_1}$, then the execution of the use case u is described as $P_u(I/t)$.

For example, suppose u is a use case with an exclusion point. Then the process algebra term for u involves a free variable X corresponding to the exclusion point, so the term is $P_u(X)$. If the extending use case u_1 has process algebra term $t = P_{u_1}$, then the execution of the use case u is described as $P_u(X/s)$ where $s = t + \epsilon$ because the execution of the extension is optional, and the empty term ϵ represents the "skip" episode which does nothing.

Note that each inclusion point and extension point occurs in only one place in the episode model, hence there is only one occurrence of the corresponding free variable in the process algebra term. Contrary to this, the placeholder for generalization can occur in one or more places in the episode model, and so substitution of the term t representing the parent episode model for the variable V representing the placeholder in the term $P_u(V)$ for the episode model of the child use case u may occur in more than one position.

The model semantics S_M can be defined based on related use case relationships. The meaning of use case relationships is analysed here at the episode level. To simplify the illustration, this analysis only deals with the situation where use case model M contains only two use cases: u_1 and u_2. For the use case model containing more than two use cases, its semantics can be defined in a similar manner because two use cases can be mapped into one use case recursively within the use case model.

Suppose use cases u_1 and u_2 interact with the same actor. Let P_{u_1} be the process algebra term of use case u_1, P_{u_2} be the process algebra term of use case u_2. Use cases u_1 and u_2 are mapped into a single use case u. Let P_u be the process algebra term of use case u. The following section analyses the semantics of a use case model with two use cases u_1 and u_2 by discussing the semantics of the two use cases when there is one relationship between them. It analyses independent use cases first, the deals with the relationships of generalization, inclusion, extension, precedence, similarity, and equivalence.

(1). No Relationship

Suppose use cases u_1 and u_2 are concrete use cases and there is no use case relationship between them, they can be mapped into a single use case u which consists of the "or" of their respective episode models.

So the semantics S_M is:

$S_M = P_u = P_{u_1} + P_{u_2}$

(2). Generalization

As described in Section 3.1.2, a *generalization* relationship between two use cases implies that the child use case contains all the sequences of behavior, including extension/inclusion points defined in the parent use case, and participates in all relations of the parent use case.

Suppose that use case u_2 is the parent of use case u_1. Let $P_{u_1}(V)$ be the semantics of u_1 with the free variable V representing the placeholder for the inherited episode model from u_2. Let $t = P_{u_2}$ be the semantics of the inherited episode model from u_2.

We can assume that u_1 is a concrete use case, otherwise we have at most a single use case (the parent) that can be executed — and nothing to analyse. There are two cases to consider: when the parent u_2 is an abstract use case, and when it is a concrete use case.

When u_2 is an abstract use case, there is only one use case that can be executed, namely the child u_1. Then

$S_M = P_{u_1}(V/t)$

When the parent u_2 is a concrete use case, then the system can execute either u_1 or u_2, so the semantics is:

$S_M = P_{u_1}(V/t) + P_{u_2}$

which is the same as P_u for a use case u that has an episode model consisting of the alternation of the episode model of u_2 and the episode model of u_1 with the placeholder replaced by u_2's episode tree.

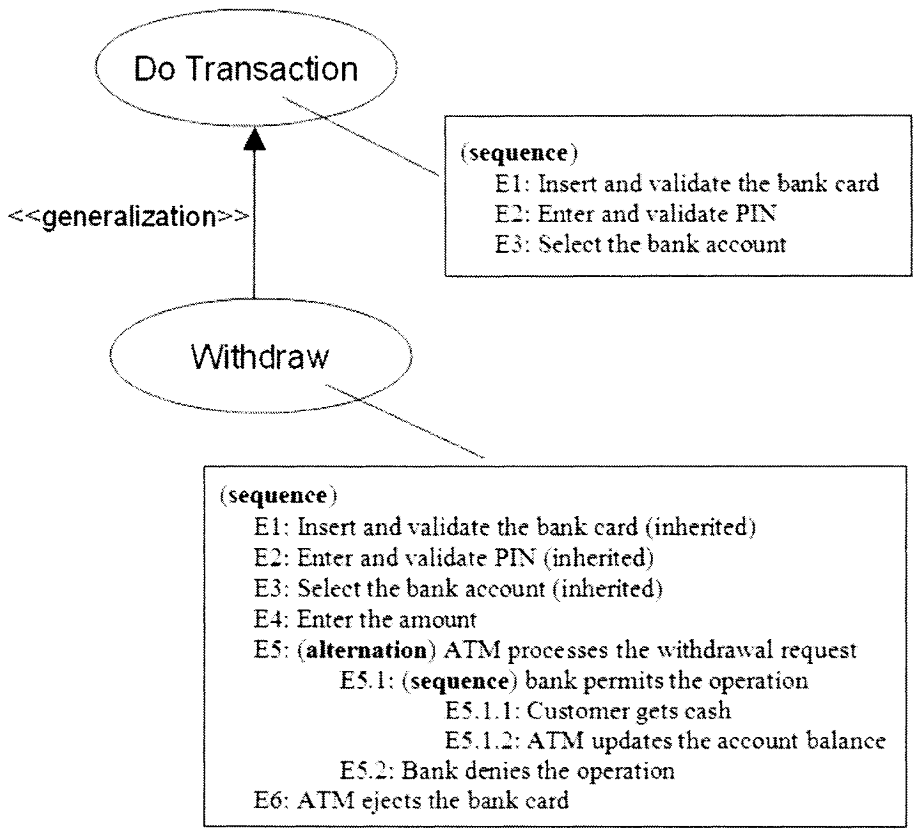
For example, in Figure 10, the use case "Do Transaction" is an abstract use case. It is the parent of the use case "Withdraw". When they are mapped into a single use case, episode E_1, E_2 and E_3 in use case "Do Transaction" are mapped to use case "Withdraw" based on their corresponding episode placeholders. Figure 11 shows the episode model for the single use case after mapping.

(3). Inclusion

As described in Section 3.1.2, an *inclusion* relationship between two use cases specifies that one use case explicitly incorporates the behavior of another at the *inclusion point.* When one use case instance reaches the *inclusion point*, it performs the behavior of the included use case and then continues.

Suppose that use case u_1 includes use case u_2. Let $P_{u_1}(I)$ be the semantics of u_1 with the free variable I representing the pseudo-episode for the included episode model from u_2. Let $t = P_{u_2}$ be the semantics of the included episode model of u_2.

We can assume that u_1 is a concrete use case, otherwise we have at most a single use case u_2 that can be executed — and nothing to analyse. There are two cases to consider: when u_2 is an abstract use case, and when it is a concrete use case.



```
(sequence)
    E1: Insert and validate the bank card
    E2: Enter and validate PIN
    E3: Select the bank account
```

```
(sequence)
    E1: Insert and validate the bank card (inherited)
    E2: Enter and validate PIN (inherited)
    E3: Select the bank account (inherited)
    E4: Enter the amount
    E5: (alternation) ATM processes the withdrawal request
            E5.1: (sequence) bank permits the operation
                    E5.1.1: Customer gets cash
                    E5.1.2: ATM updates the account balance
            E5.2: Bank denies the operation
    E6: ATM ejects the bank card
```

Figure 10: Example: *Generalization* Relationship

```
(sequence)
    E1: Insert and validate the bank card
    E2: Enter and validate PIN
    E3: Select the bank account
    E4: Enter the amount
    E5: (alternation) ATM processes the withdrawal request
            E5.1: (sequence) bank permits the operation
                    E5.1.1: Customer gets cash
                    E5.1.2: ATM updates the account balance
            E5.2: Bank denies the operation
    E6: ATM ejects the bank card
```

Figure 11: Example: Mapping for the *Generalization* Relationship

When u_2 is an abstract use case, there is only one use case that can be executed, namely u_1. Then

$S_M = P_{u_1}(I/t)$

When the parent u_2 is a concrete use case, then the system can execute either u_1 or u_2, so the semantics is:

$S_M = P_{u_1}(I/t) + P_{u_2}$

which is the same as P_u for a use case u that has an episode model consisting of the alternation of the episode model of u_2 and the episode model of u_1 with the pseudo-episode replaced by u_2's episode tree.

For example, in Figure 12, the use case "Withdraw" includes the use case "PIN Validation". Use case "PIN Validation" is an abstract use case. When they are mapped into a single use case, episode E_2 and its primitive episodes in use case "PIN Validation" are inserted into the inclusion point (E_2) in use case "Withdraw" directly. Figure 13 shows the episode model for the single use case after mapping.

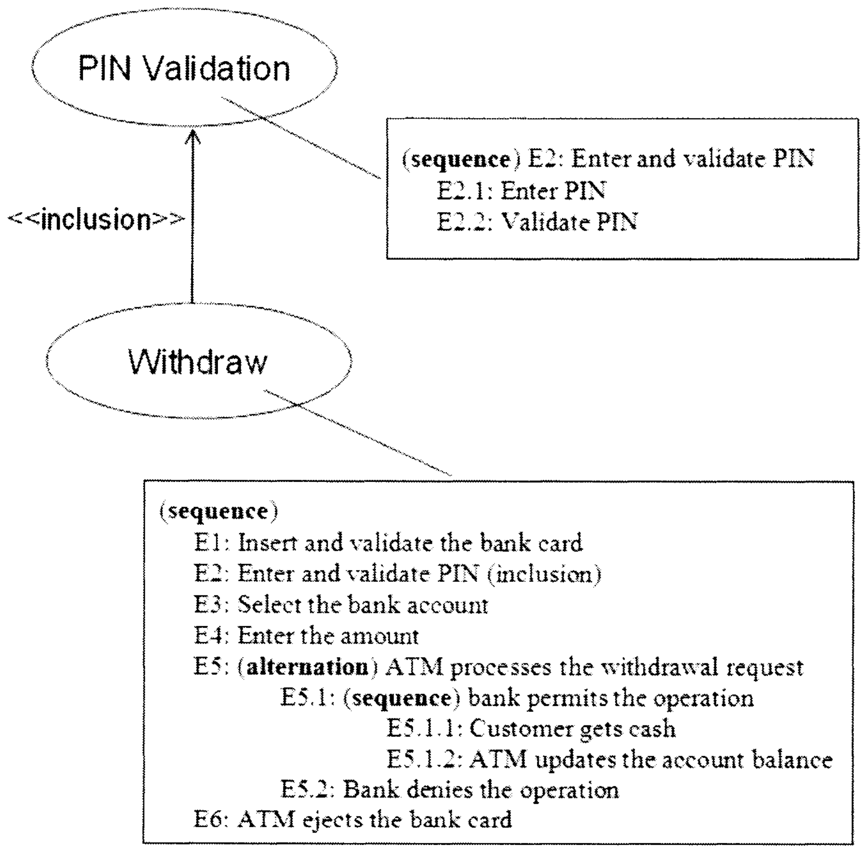

PIN Validation
<<inclusion>>
Withdraw
(sequence) E2: Enter and validate PIN
E2.1: Enter PIN
E2.2: Validate PIN
(sequence)
E1: Insert and validate the bank card
E2: Enter and validate PIN (inclusion)
E3: Select the bank account
E4: Enter the amount
E5: (alternation) ATM processes the withdrawal request
E5.1: (sequence) bank permits the operation
E5.1.1: Customer gets cash
E5.1.2: ATM updates the account balance
E5.2: Bank denies the operation
E6: ATM ejects the bank card

Figure 12: Example: *Inclusion* Relationship

(sequence)
- E1: Insert and validate the bank card
- E2: **(sequence)** Enter and validate PIN
    - E2.1: Enter PIN
    - E2.2: Validate PIN
- E3: Select the bank account
- E4: Enter the amount
- E5: **(alternation)** ATM processes the withdrawal request
    - E5.1: **(sequence)** bank permits the operation
        - E5.1.1: Customer gets cash
        - E5.1.2: ATM updates the account balance
    - E5.2: Bank denies the operation
- E6: ATM ejects the bank card

Figure 13: Example: Mapping for the *Inclusion* Relationship

(4). Extension

As described in Section 3.1.2, an *extension* relationship between two use cases specifies that one use case extends the behavior of another at the given *extension point.* One use case extends another by introducing alternative or exceptional processes. The extension relationship is typically used in exceptional circumstances.

Suppose that use case u_1 is extended by use case u_2. Let $P_{u_1}(X)$ be the semantics of u_1 with the free variable X representing the pseudo-episode for the extending episode model from u_2. Let $t = P_{u_2}$ be the semantics of the extending episode model of u_2. Let $s = t + \epsilon$.

We can assume that u_1 is a concrete use case, otherwise we have at most a single use case u_2 that can be executed — and nothing to analyse. There are two cases to consider: when u_2 is an abstract use case, and when it is a concrete use case.

When u_2 is an abstract use case, there is only one use case that can be executed, namely u_1. Then

$S_M = P_{u_1}(X/s)$

When the parent u_2 is a concrete use case, then the system can execute either u_1 or u_2, so the semantics is:

$S_M = P_{u_1}(X/s) + P_{u_2}$

which is the same as P_u for a use case u that has an episode model consisting of the alternation of the episode model of u_2 and the episode model of u_1 with the pseudo-episode replaced by the alternation of "skip" and the episode tree of u_2.

For example, in Figure 14, the use case "Resume PIN Entry" extends the use case "Withdraw". When they are mapped into a single use case, episode E_2 and its child episodes in use case "Resume PIN Entry" are inserted into the extension point (E_2) in use case "Withdraw" directly. Figure 15 shows the episode model for the single use case after mapping.

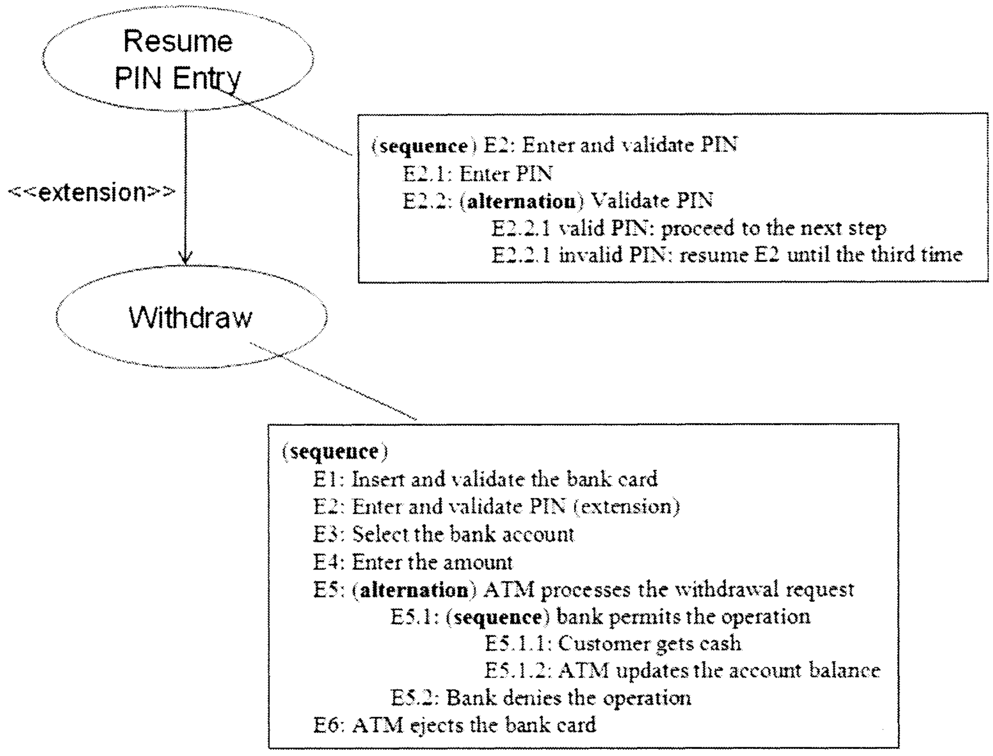

Resume PIN Entry
<<extension>>
Withdraw
(sequence) E2: Enter and validate PIN
E2.1: Enter PIN
E2.2: (alternation) Validate PIN
E2.2.1 valid PIN: proceed to the next step
E2.2.1 invalid PIN: resume E2 until the third time
(sequence)
E1: Insert and validate the bank card
E2: Enter and validate PIN (extension)
E3: Select the bank account
E4: Enter the amount
E5: (alternation) ATM processes the withdrawal request
E5.1: (sequence) bank permits the operation
E5.1.1: Customer gets cash
E5.1.2: ATM updates the account balance
E5.2: Bank denies the operation
E6: ATM ejects the bank card

Figure 14: Example: *Extension* Relationship

```
(sequence)
    E1: Insert and validate the bank card
    E2: (sequence) Enter and validate PIN
            E2.1: Enter PIN
            E2.2: (alternation) Validate PIN
                    E2.2.1 valid PIN: proceed to the next step
                    E2.2.1 invalid PIN: resume E2 until the third time
    E3: Select the bank account
    E4: Enter the amount
    E5: (alternation) ATM processes the withdrawal request
            E5.1: (sequence) bank permits the operation
                    E5.1.1: Customer gets cash
                    E5.1.2: ATM updates the account balance
            E5.2: Bank denies the operation
    E6: ATM ejects the bank card
```

Figure 15: Example: Mapping for the *Extension* Relationship

(5). Precedence

Suppose there is a *precedence* relationship between use case u_1 and u_2. Use case u_1 precedes to u_2. When u_1 is concrete and u_2 is abstract then the semantics S_M is:

$$S_M = P_u = P_{u_1} \cdot P_{u_2}$$

When u_2 is concrete too, then semantics S_M is:

$$S_M = P_u = (P_{u_1} \cdot P_{u_2}) + P_{u_2}$$

Let E_1 be the episode model of u_1 and E_2 the episode model of u_2. This is the same semantics as a use case u where the episode model is $Sequence[E_1, E_2]$ in the first case, and $Alternation[Sequence[E_1, E_2], E_2]$ in the second case.

(6). Similarity

Suppose there is a *similarity* relationship between use cases u_1 and u_2. Let E_1 be the episode model of u_1 and E_2 the episode model of u_2. It means that there is some association between use cases, but the association is not further specified.

We treat this case as if u_1 and u_2 were independent:

$$S_M = P_u = P_{u_1} + P_{u_2}$$

where u is a use case with episode model $Alternation[E_1, E_2]$.

(7). Equivalence

Suppose there is an *equivalence* relationship between use cases u_1 and u_2. Let E_1 be the episode model of u_1 and E_2 the episode model of u_2. The equivalence relationship requires that E_1 and E_2 be semantically equivalent, so $P_{u_1} = P_{u_2}$. Hence,

$$S_M = P_u = P_{u_1} = P_{u_2} = P_{u_1} + P_{u_2}$$

where u is a use case with episode model $Alternation[E_1, E_2]$ which simplifies to E_1.

3.3 Properties and Behavior Preservation

As introduced in Section 2.1, Opdyke identifies seven invariants required to preserve the behavior of C++ program. When a refactoring tends to violate an invariant, enabling conditions are added to ensure that the invariant is preserved.

Similarly, eight invariants are identified in this research to preserve the behavior of the use case model built from the metamodel in Section 3.1.

1. **Unique Parent:** It supports only single inheritance systems, without cycles in the inheritance. Therefore, a use case, actor, goal, task, and episode must always have at most one direct parent and its parent must not also be one of its children.

2. **Distinct Environmental Entity Name:** Each entity at the environment level must have a unique name, which includes use case, actor, user, task, service, and goal. The scope of each entity is the entire use case model.

3. **Distinct Episode Name:** Each episode must have a unique name. This allows the same episode to be reused among different use cases within the same use case model. It avoids redundant definition for the same episode.

4. **Distinct Event Name:** Each event must have a unique name. This allows the same event to be reused among different episodes within the same use case model. It avoids the redundant definition for the same event.

5. **Inherited Episode/Event not Redefined:** An episode inherited from the parent use case cannot be redefined in any of its child use cases. Child use cases may contain an episode placeholder for this episode. Similarly, an event inherited from a parent episode cannot be redefined in any of its child episodes. Child episodes may contain an event placeholder for this event.

6. **Equivalent References to Goals:** If the goal model is defined in the use case model, the association relationship between goals and concrete actors, services must be preserved. This ensures the consistency between the goal model and the core use case model (the interaction between use cases and actors).

7. **Equivalent References to Tasks:** If the task model is defined in the use case model, the association relationship between tasks and users, services must be preserved. This ensures the consistency between the task model and the core use case model (the interaction between use cases and actors).

8. **Semantically Equivalence:** The use case model before and after a refactoring must be semantically equivalent. The semantics of the use case model is the interaction between use cases and actors within the model. This allows for several useful changes that do not affect equivalence:

   - Use case, actor, user, service, goal, task, episode and event can be added or removed if they are isolated from the rest elements in the use case model.
   - Redundant use case, actor, and episode can be removed to simplify the use case model.
   - Use case, actor, user, service, goal, task, episode, and event can be given a new name.

In the refactoring definition, the behavior preservation is argued in terms of these syntactic and semantic properties of the use case model. To simplify the illustration of the refactoring definition, the thesis assumes that the whole system is modeled as a single *service* except for three refactorings: *Create_Empty_Service*, *Delete_Unreferenced_Service*, and *Change_Service_Name*. A *service* is a package of functional entities (features) offered to the users in order to satisfy one or more goals that the users have. It does not affect the interaction between actors and use cases. Therefore, this simplification does not affect the correctness of the refactoring definition.

3.4 Use Case Refactorings

This section presents an overview on use case refactorings. There are fifty-three refactorings in total, including thirty-eight at the environment level, twelve at the structure level, and three at the event level. These refactorings are designed to be simple so that it is easier to show that they are behavior preserving. For completeness, the definition of each refactoring details its description, arguments, preconditions, postconditions and argues that it is behavior preserving. Since this information provides a level of detail beyond what may be of interest to some readers, it is presented in the Appendix. This section also presents a refactoring template and illustrates it using a refactoring example.

3.4.1 List of Use Case Refactorings

The complete list of use case refactorings is showed as follows. These refactorings are organized according to different levels. There are different categories at each level.

A. Refactorings at the Environment Level

A.1 Create an Environmental Entity: This category contains refactorings to create new environmental entities: use case, actor, user, service, goal and task.

A.1.1 Create_Empty_Usecase

A.1.2 Create_Empty_Actor

A.1.3 Create_Empty_User

A.1.4 Create_Empty_Service

A.1.5 Create_Empty_Goal

A.1.6 Create_Empty_Task

A.2 Delete an Environmental Entity: This category contains refactorings to delete unreferenced environmental entities: use case, actor, user, service, goal and task.

A.2.1 Delete_Unreferenced_Usecase

A.2.2 Delete_Unreferenced_Actor

A.2.3 Delete_Unreferenced_User

A.2.4 Delete_Unreferenced_Service

A.2.5 Delete_Unreferenced_Goal

A.2.6 Delete_Unreferenced_Task

A.3 Change an Environmental Entity: This category contains refactorings to change the name for environmental entities: use case, actor, user, service, goal and task.

A.3.1 Change_Usecase_Name

A.3.2 Change_Actor_Name

A.3.3 Change_User_Name

A.3.4 Change_Service_Name

A.3.5 Change_Goal_Name

A.3.6 Change_Task_Name

A.4 Compose an Environmental Entity: This category contains refactorings to compose use cases and actors with different relationships. Each refactoring below is listed with a brief description.

A.4.1 Change_Parent_Usecase: change the parent of a use case to another one.

A.4.2 Change_Parent_Actor: change the parent of an actor to another one.

A.4.3 Usecase_Generalization_Merger: merge two use cases with the generalization relationship into one use case.

A.4.4 Usecase_Inclusion_Merger: merge two use cases with the inclusion relationship into one use case.

A.4.5 Usecase_Extension_Merger: merge two use cases with the extension relationship into one use case.

A.4.6 Usecase_Precedence_Merger: merge two use cases with the precedence relationship into one use case.

A.4.7 Usecase_Equivalence_Generation: generate an equivalence relationship between two use cases if they are equivalent.

A.4.8 Reuse_Usecase: keep only one use case and remove the other when two use cases are equivalent.

A.4.9 Merge_Usecases: merge two use cases into one use case.

A.4.10 Generalize_Usecases: generate a parent use case for a list of use cases.

A.4.11 Merge_Actors: merge two actors into one actor.

A.4.12 Generalize_Actors: generate a parent actor for a list of actors.

A.5 Decompose an Environmental Entity: This category contains refactorings to decompose use cases and actors. Each refactoring below is listed with a brief description.

A.5.1 Split_Usecase: split one use case into two use cases and there is no use case relationship between these two use cases.

A.5.2 Usecase_Generalization_Generation: split one use case into two use cases and generate a generalization relationship between these two use cases.

A.5.3 Usecase_Inclusion_Generation: split one use case into two use cases and generate an inclusion relationship between these two use cases.

A.5.4 Usecase_Extension_Generation: split one use case into two use cases and generate an extension relationship between these two use cases.

A.5.5 Usecase_Precedence_Generation: split one use case into two use cases and generate a precedence relationship between these two use cases.

A.5.6 Split_Actor: split one actor into two actors.

A.5.7 Actor_Generalization_Generation: split one actor into two actors and generate a generalization relationship between these two actors.

A.6 Move an Environmental Entity: This category contains a refactoring that moves a goal from child actors to their parent actor.

A.6.1 Move_Goal_To_Parent_Actor

B. Refactorings at the Structure Level

B.1 Create a Structural Entity: This category contains a refactoring to create a new episode.

B.1.1 Create_Empty_Episode

B.2 Delete a Structural Entity: This category contains a refactoring to delete an unreferenced episode.

B.2.1 Delete_Unreferenced_Episode

B.3 Change a Structural Entity: This category contains a refactoring to change the name of an episode.

B.3.1 Change_Episode_Name

B.4 Compose a Structural Entity: This category contains refactorings to compose a primitive episode with different episode relationships. Each refactoring below is listed with a brief description.

B.4.1 Episode_Sequence_Merger: merge two primitive episodes with the sequence relationship into one primitive episode.

B.4.2 Episode_Alternation_Merger: merge two primitive episodes with the alternation relationship into one primitive episode.

B.4.3 Episode_Parallel_Merger: merge two primitive episodes with the parallel relationship into one primitive episode.

B.4.4 Generalize_Episodes: generate a parent episode for a list of episodes.

B.5 Decompose a Structural Entity: This category contains refactorings to decompose a primitive episode and generate different episode relationships. Each refactoring below is listed with a brief description.

B.5.1 Episode_Sequence_Generation: split one primitive episode into two episodes and generate a sequence relationship between these two episodes.

B.5.2 Episode_Alternation_Generation: split one primitive episode into two episodes and generate an alternation relationship between these two episodes.

B.5.3 Episode_Parallel_Generation: split one primitive episode into two episodes and generate a parallel relationship between these two episodes.

B.6 Move a Structural Entity: This category contains a refactoring that moves an episode from child use cases to their parent use case.

B.6.1 Move_Episode_To_Parent_Usecase

B.7 Convert a Structural Entity: This category contains a refactoring to convert an episode into a use case.

B.7.1 Convert_Episode_To_Usecase

C. Refactorings at the Event Level

C.1 Create an Event: This category contains a refactoring to create a new event.

C.1.1 Create_Unreferenced_Event

C.2 Delete an Event: This category contains a refactoring to delete an unreferenced event.

C.2.1 Delete_Unreferenced_Event

C.3 Change an Event: This category contains a refactoring to change the name of an event.

C.3.1 Change_Event_Name

3.4.2 Notations and Functions for Defining Use Case Refactorings

This section introduces a list of notations and functions used to define use case refactorings. Attributes are referenced using a *dot* notation. For example, all actors within the use case model is referenced by *Model.actors*. This section only lists those attributes used for defining use case refactorings. The post state is expressed by prime ($'$). For example, *Model′* is the use case model after the refactoring where Model is the use case model before the refactoring.

Notations for Defining Use Case Refactorings

1. {}: the empty set.

2. **environmental entity**: It refers to actor, use case, goal, task, user, and service.

3. **element**: It refers to environmental entity, episode and event.

4. **referenced/unreferenced**: If the element v is referenced by the element a, the element v can be accessed in a by $a.v$. If the element v is unreferenced by the element a, v cannot be accessed in a by the *dot* notation directly.

5. **association relationship**: If the actor interacts with the use case directly, there is an association relationship between the actor and the use case.

6. **use**: It refers that the actor communicates with the use case when there is the association relationship between the actor and the use case.

7. **Model**: the use case model conformed to the use case metamodel defined in Section 3.1. It has the following attributes:

   (1). **usecases**: the set of use cases in the use case model.

   (2). **actors**: the set of actors in the use case model.

   (3). **users**: the set of users in the use case model.

   (4). **tasks**: the set of tasks in the use case model.

   (5). **goals**: the set of goals in the use case model.

   (6). **services**: the set of services in the use case model.

   (7). **episodes**: the set of episodes in the use case model.

   (8). **events**: the set of events in the use case model.

8. **use case**: It has the following attributes:

   (1). **name**: the name of the use case. The type of the name is string.

   (2). **type**: the type of the use case. The set of types is {abstract,concrete}, which refer to the abstract use case and concrete use case respectively.

   (3). **episodes**: the set of episodes in the use case. The episode can be primitive episode, composite episode or pseudo-episode. These episodes are episodes participating in the episode model of the use case. They do not include inherited episodes, which are represented by the episode placeholder in the use case.

   (4). **inheritedEpisodes**: the set of episodes that are inherited from the parent use case. The episode can be primitive episode, composite episode or pseudo-episode.

   (5). **actors**: the set of actors in the use case model that interact with the use case directly.

   (6). **usecases**: the set of use cases in the use case model that have use case relationships with the use case.

   (7). **parentUsecase**: the direct parent use case. One use case can have multiple parent use cases, but it can have only one direct parent.

   (8). **rootEpisode**: the root episode for the root node of the episode tree. One use case has at most one root episode.

(9). **episodeTree**: the episode tree of the use case.

9. **actor**: It has the following attributes:

   (1). **name**: the name of the actor. The type of the name is string.

   (2). **type**: the type of the actor. The set of types is {abstract, concrete}, which refer to the abstract actor and concrete actor respectively.

   (3). **actors**: the set of actors in the use case model that have the direct actor relationship with the actor.

   (4). **users**: the set of users in the use case model that are instances of the actor.

   (5). **goals**: the set of goals of the actor.

   (6). **usecases**: the set of use cases that interact with the actor directly.

   (7). **inheritedUsecases**: the set of use cases that have the explicitly specified association relationship with parent actors of the actor. The actor inherits these association relationships from parent actors.

   (8). **parentActor**: the direct parent actor. One actor can have multiple parent actors, but it can have only one direct parent.

10. **user**: It has the following attributes:

    (1). **name**: the name of the user. The type is string.

    (2). **actors**: the set of actors in the use case model that the user is the instance of.

    (3). **tasks**: the set of tasks in the use case model that the user has.

    (4). **usecases**: the set of use cases that the user participates in.

11. **service**: it has the following attributes:

    (1). **name**: the name of the service. The type of the name is string.

    (2). **usecases**: the set of use cases in the use case model that describes the service.

    (3). **tasks**: the set of tasks in the use case model that fulfills the service.

    (4). **goals**: the set of goals in the use case model that satisfies the service.

12. **task**: It has the following attributes:

    (1). **name**: the name of the task. The type of the name is string.

    (2). **users**: the set of users in the use case model that have the task.

(3). **tasks**: the set of tasks in the use case model that have the generalization relationship or inclusion relationship with the task.

13. **goal**: It has the following attributes:

   (1). **name**: the name of the goal. The type of the name is string.

   (2). **actors**: the set of actors in the use case model that have the goal.

   (3). **services**: the set of services in the use case model that the goal is satisfied to.

   (4). **goals**: the set of goals in the use case model that have the generalization relationship or inclusion relationship with the goal.

14. **episode**: It has the following attributes:

   (1). **name**: the name of the episode. The type of the name is string.

   (2). **type**: the type of the episode. The set of types is { Primitive Episode, Sequence Episode, Alternation Episode, Iteration Episode, Parallel Episode, Pseudo-Episode }.

   (3). **usecases**: the set of use cases that have the episode.

   (4). **episodes**: the set of episodes that directly interact with the episode.

   (5). **events**: the set of events that the episode has.

   (6). **parentEpisode**: the parent of the episode.

   (7). **eventTree**: the event tree of the episode.

15. **event**: It has the following attributes:

   (1). **name**: the name of the event. The type of the name is string.

   (2). **type**: the type of the event. The set of types is { Stimulus, Response, Action }.

   (3). **episodes**: the set of episodes that have the event.

   (4). **events**: the set of events that directly interact with the event.

16. **episode tree**: It has the following attribute:

   (1). **episodes**: the set of episodes that occur in the episode tree.

17. **event tree**: It has the following attribute:

   (1). **events**: the set of events that occur in the event tree.

18. **Precedence** $[u_1, u_2]$: where u_1 and u_2 are use cases. This notation refers that there is a *precedence* relationship between use case u_1 and u_2. Use case u_1 precedes to use case u_2.

19. **Inclusion** $[u_1, u_2]$: where u_1 and u_2 are use cases. This notation refers that there is an *inclusion* relationship between use case u_1 and u_2. Use case u_1 includes use case u_2.

20. **Extension** $[u_1, u_2]$: where u_1 and u_2 are use cases. This notation refers that there is an *extension* relationship between use case u_1 and u_2. Use case u_2 extends use case u_1.

21. **Equivalence** $[u_1, u_2]$: where u_1 and u_2 are use cases. This notation refers that there is an *equivalence* relationship between use case u_1 and u_2.

22. **Sequence** $[e_1, e_2]$: where e_1 and e_2 are episodes. This notation refers that there is a *sequence* relationship between episode e_1 and e_2. Episode e_2 follows episode e_1 immediately.

23. **Alternation** $[e_1, e_2]$: where e_1 and e_2 are episodes. This notation refers that there is an *alternation* relationship between episode e_1 and e_2.

24. **Parallel** $[e_1, e_2]$: where e_1 and e_2 are episodes. This notation refers that there is a *parallel* relationship between episode e_1 and e_2.

25. **Pseudo** $[u]$: where u is the use case. This notation refers to the pseudo-episode indicating the inclusion point or extension point. Use case u is the included use case or extending use case.

26. **Placeholder** $[x]$: where x is an episode or an event. This notation refers to the episode placeholder or event placeholder for x.

27. P_x: the process algebra term of x where x can be a use case, episode or event.

28. $T[e]$: the episode tree of the episode e where e is an episode or a list of episodes with operators: *sequence* $(\cdot)$, *alternation* $(+)$, *parallel* $(||)$, *iteration* ($(x)^i$ where x is an *iteration* episode and i is the number of repetitions or an interval). This notation refers to an episode tree or episode subtree. For example, $T[e_1 \cdot e_2]$ represents an episode subtree with episode e_1 and e_2. There is a *sequence* relationship between episode e_1 and e_2. Episode e_2 follows e_1 immediately.

29. $T[v]$: the event tree of the event v where v is an event or a list of events with operators: *sequence* $(\cdot)$, *alternation* $(+)$, *parallel* $(||)$, *iteration* ($(\mathrm{x})^i$ where x is an event or a list of events with operators and i is the number of repetitions or an interval). This notation refers to an event tree or event subtree. For example, $T[v_1 \cdot v_2]$ represents an event tree or event subtree with event v_1 and v_2. There is a *sequence* relationship between event v_1 and v_2. Event v_2 follows v_1 immediately.

Functions for Defining Use Case Refactorings

1. **DeleteUnreferencedEpisode(e)**: where e is an unreferenced episode. This function applies refactoring *Delete_Unreferenced_Episode* to the episode e.

2. **DeleteUnreferencedUser(u)**: where u is an unreferenced user. This function applies refactoring *Delete_Unreferenced_User* to the user u.

3. **DeleteUnreferencedGoal(g)**: where g is an unreferenced goal. This function applies refactoring *Delete_Unreferenced_Goal* to the goal g.

4. **DeleteUnreferencedTask(t)**: where t is an unreferenced task. This function applies refactoring *Delete_Unreferenced_Task* to the task t.

5. **SubstituteEpisode(e_1, e_2)**: where e_1 and e_2 are episodes in the use case model. This function substitutes every occurrence of the episode e_1 in the use case model with e_2.

6. **SubstituteEpisodeWithPlaceholder(u, e)**: where e is an episode in the use case u. This function substitutes every occurrence of the episode e in the episode tree of the use case u with Placeholder [e].

7. **SubstitutePlaceholderWithEpisode(u, *Placeholder* [e])**: where Placeholder [e] is an episode placeholder in the use case u. This function substitutes every occurrence of Placeholder [e] with the corresponding episode e in the episode tree of the use case u.

8. **SubstituteEpisodeWithPseudo(u_1, e, *Pseudo* [u_2])**: where e is an episode in the use case u_1. This function establishes an inclusion relationship between use cases u_1 and u_2. The use case u_1 includes the use case u_2. This function substitutes the episode e in the episode tree of the use case u_1 with the pseudo-episode Pseudo [u_2].

9. **SubstituteEpisodeTreeInUsecase(u, T_1, T_2)**: where u is a use case, T_1 and T_2 is the episode tree or episode subtree in the use case u. This function substitutes every occurrence of T_1 in the episode tree of the use case u with T_2.

10. **DeleteEpisodeTreeInUsecase(u, T_1)**: where u is a use case, T_1 is the episode subtree in the use case u. This function deletes every occurrence of T_1 in the episode tree of the use case u.

11. **SubstituteEventWithPlaceholder(e, v)**: where v is an event in the episode e. This function substitutes every occurrence of the event v in the event tree of the episode e with Placeholder [v].

12. **IsEpisodeSubTree(T, t)**: where T is the episode tree of a use case and t is an episode tree. This function returns true if t is an episode subtree of the episode tree T. Otherwise, it returns false.

13. **IsEventSubTree**(T, t): where E is the event tree of an episode and t is an event tree. This function returns true if t is an event subtree of the event tree T. Otherwise, it returns false.

3.4.3 Refactoring Template

A template is designed to define each use case refactoring. It is described below:

- Description: It presents an overview of the refactoring, including the motivation and how the refactoring works.
- Arguments: It lists input arguments of the refactoring.
- Preconditions: It specifies conditions that must be satisfied before the refactoring can be applied. All preconditions must be met.
- Postconditions: It specifies the final state of the use case model after the refactoring is applied. All postconditions must be met.
- Verification: It discusses preconditions and postconditions in terms of the behavior preservation. It verifies if the refactoring preserves the behavior of the use case model.

As an example, Figure 16 and Figure 17 show how the *Usecase_Inclusion_Generation* refactoring is defined using this template.

Description: This refactoring splits one use case into two and generates an inclusion relationship between these two use cases. When the episode model of a use case is too complicated, it is preferable to split the episode tree. After the refactoring, a new use case is created with an episode subtree of the original use case. The original use case includes the new use case. This refactoring helps manage the use case granularity, reduce the redundancy, and improve the understandability of the use case model.

Arguments: use case U, the episode tree T of the use case U, the episode subtree t_1 for creating the new use case, string newUsecaseName.

Preconditions:

(a). The episode subtree t_1 is a subtree of T.

$$U \in Model.usecases \wedge U.episodeTree = T \wedge IsEpisodeSubTree(T, t_1)$$

(b). The name of the new use case does not clash with an already existing use case within the use case model.

$$\forall usecase \in Model.usecases, usecase.name \neq newUsecaseName.$$

Postconditions:

(a). An abstract use case U_1 is created with the name of newUsecaseName. The episode tree of the new use case U_1 is the episode subtree t_1. The use case U includes the new use case U_1.

$$\exists U_1 \notin Model.usecases \wedge U_1 \in Model'.usecases$$
$$\wedge\ U_1.type = abstract \wedge U_1.episodeTree = t_1 \wedge Inclusion[U, U_1]$$

(b). The episode tree t_1 is substituted by a pseudo-episode in the episode tree T. The pseudo-episode indicates the inclusion point for the inclusion relationship between use cases U and U_1.

$$SubstituteEpisodeTreeInUsecase(U, t_1, T[Pseudo[U_1]])$$

Figure 16: Example: *Usecase_Inclusion_Generation* Refactoring - Part (a)

Verification: Conversely to the *Usecase_Inclusion_Merger* refactoring, this refactoring splits one use case into two and generates an inclusion relationship between these two use cases. The semantics of the inclusion relationship is analysed in Section 3.2.2. The new use case U_1 is an abstract use case. It cannot be instantiated on its own. The use case U' includes the behavior of the new use case U_1 at the inclusion point. The interaction between the use case U and related actors is preserved. Therefore, this refactoring is a behavior preserving transformation.

Precondition (a) is obvious. Precondition (b) ensures distinct environmental entity name (invariant two in Section 3.3).

Postcondition (a) ensures that the new use case U_1 is an abstract use case. Therefore, the behavior of the use case model with use cases U' and U_1 can be verified based on the interaction between the use case U' and related actors. Postcondition (b) ensures that the episode tree T is adjusted properly after the refactoring by replacing the episode subtree t_1 with the inclusion point.

Figure 17: Example: *Usecase_Inclusion_Generation* Refactoring - Part (b)

Chapter 4

A Tool for Use Case Modeling and Refactoring

One of important goals of this research is to demonstrate that refactoring use case models is feasible and practical. As part of this research, we develop a tool for use case modeling and refactoring. The tool enables us to build the use case model based on the metamodel defined in Section 3.1. We can evolve the model by applying use case refactorings under the tool support. The tool allows us to test many of our refactoring ideas. It gives us an insight into the nature of the use case evolution and how and when to apply refactorings.

The tool is developed with three master students: Jian Xu, Wei Yu and Renhong Luo. Some work is described in their master theses [XU04] [YU04]. Basically, Jian Xu mainly works on the environment level. He also integrates different functional modules together. Wei Yu focuses on the structure level and event level. She defines the episode model and some use case refactorings. Renhong Luo works on the goal module, which includes goal definition, data display and refactoring supporting module. I manage the whole project and supervise the team. I provide them with detailed definitions of use case refactorings. We work closely to design the system and to implement refactorings. For the completeness of the thesis, I introduce the tool as a whole system here.

4.1 Drawlets Framework

Drawlets [ROLE] is a two-dimensional graphics framework written in Java for building graphical applications. Figure 18 shows the class diagram for the intended use of Drawlets fundamental roles. *DrawingCanvas* contains the *drawing* and *tool* components. The *drawing* component holds *figures* and *listeners*. The *tool* component allows user to interact with *figures* to modify their attributes: size, location, etc. A special selection *tool* can select multiple *figures* and modify all of them.

Drawlets supports many shapes: lines, freehand lines, triangles, rectangles, rounded rectangles, pentagons, polygons, ellipses, text boxes, etc. Each figure has a separate drawing tool.

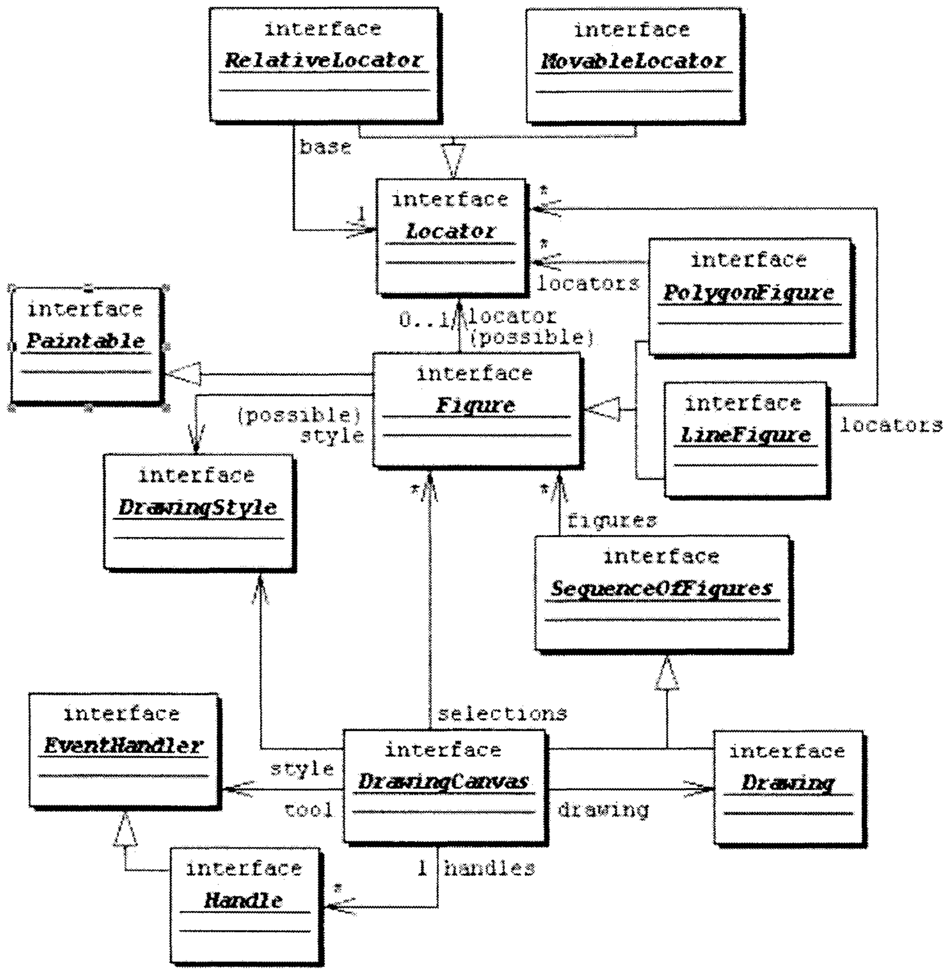


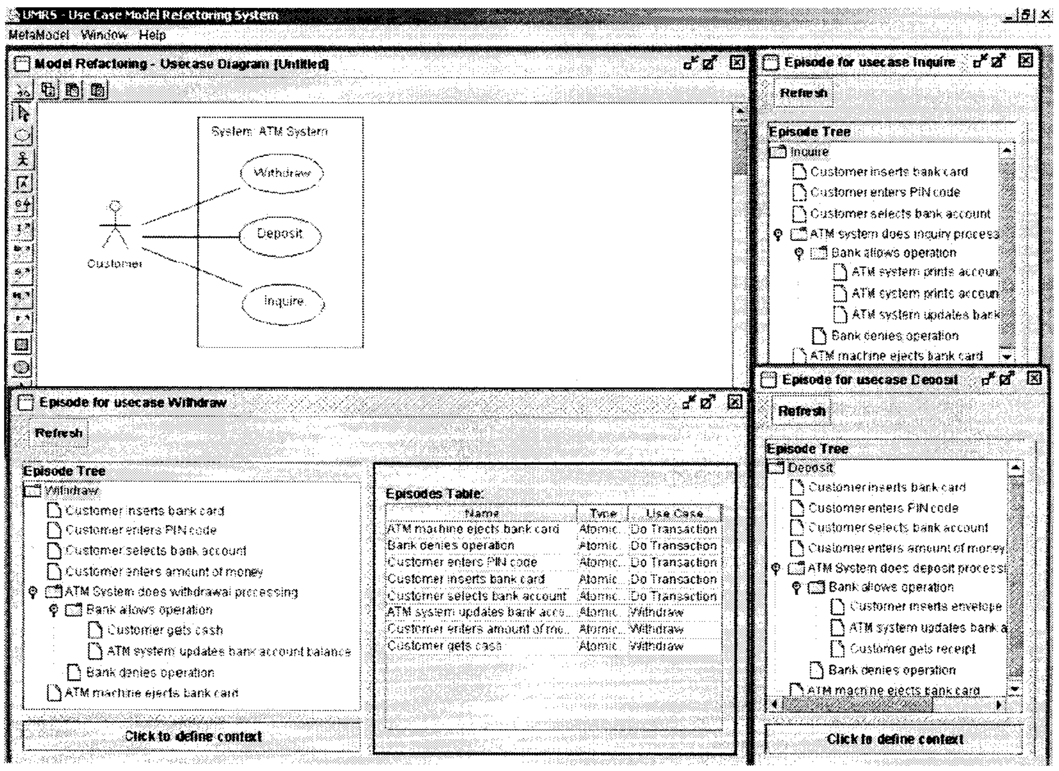
Figure 18: Class Diagram for the Intended Use of Drawlets Fundamental Roles

In this refactoring tool, we extend the Drawlets framework to define elements at the environment level. Drawlets allows us to build the tool quickly and it provides the flexibility for further modifications and extensions. For elements at the structure level and event level, we use tree and list structures to show episodes and events instead of graphical elements provided by Drawlets.

4.2 System Overview

Figure 19 shows a screen shot of the current refactoring tool. The use case model at the environment level is shown at the top-left. The structure of each use case is shown in "Episode Tree" windows respectively. The "Episodes Table" window displays a unique list of current episodes.

The tool is an MDI (Multiple Document Interface) application. The key functionality of the tool


UMRS - Use Case Model Refactoring System
MetaModel Window Help
Model Refactoring - Usecase Diagram [Untitled]
System: ATM System
Withdraw
Deposit
Inquire
Customer
Episode for usecase Inquire
Refresh
Episode for usecase Deposit
Episode for usecase Withdraw
Click to define context

Episode Tree

```
Withdraw
    Customer inserts bank card
    Customer enters PIN code
    Customer selects bank account
    Customer enters amount of money
  ATM System does withdrawal processing
      Bank allows operation
          Customer gets cash
          ATM system updates bank account balance
        Bank denies operation
    ATM machine ejects bank card
```

Episodes Table:

Name	Type	Use Case
ATM machine ejects bank card	Atomic	Do Transaction
Bank denies operation	Atomic	Do Transaction
Customer enters PIN code	Atomic	Do Transaction
Customer inserts bank card	Atomic	Do Transaction
Customer selects bank account	Atomic	Do Transaction
ATM system updates bank acco...	Atomic	Withdraw
Customer enters amount of mo...	Atomic	Withdraw
Customer gets cash	Atomic	Withdraw

Episode Tree

```
Inquire
    Customer inserts bank card
    Customer enters PIN code
    Customer selects bank account
  ATM system does inquiry processi
      Bank allows operation
          ATM system prints accoun
          ATM system prints accoun
          ATM system updates bank
        Bank denies operation
    ATM machine ejects bank card
```

Episode Tree

```
Deposit
    Customer inserts bank card
    Customer enters PIN code
    Customer selects bank account
    Customer enters amount of money
  ATM System does deposit processi
      Bank allows operation
          Customer inserts envelope
          ATM system updates bank a
          Customer gets receipt
        Bank denies operation
    ATM machine ejects bank card
```

Figure 19: A Screen Shot of the Tool

is to build the use case model and then apply use case refactorings to evolve it. Figure 20 shows the system architecture. Basically, the tool is built based on the Drawlets framework. The use case model is saved as XML files. There is separate module for each level as defined in the use case metamodel.

The system can be divided into two subsystems: Definition subsystem and Refactoring subsystem.

4.2.1 Definition Subsystem

The Definition subsystem is used to define the use case model. The use case model is constructed based on the metamodel in Section 3.1. The definition data is stored in the XML format. The environment level is shown in graphics elements supported by Drawlets while the other two levels are shown using tree and list structures. Elements at the environment level, such as use case and actor, are extended from the basic shapes defined in the Drawlets framework.

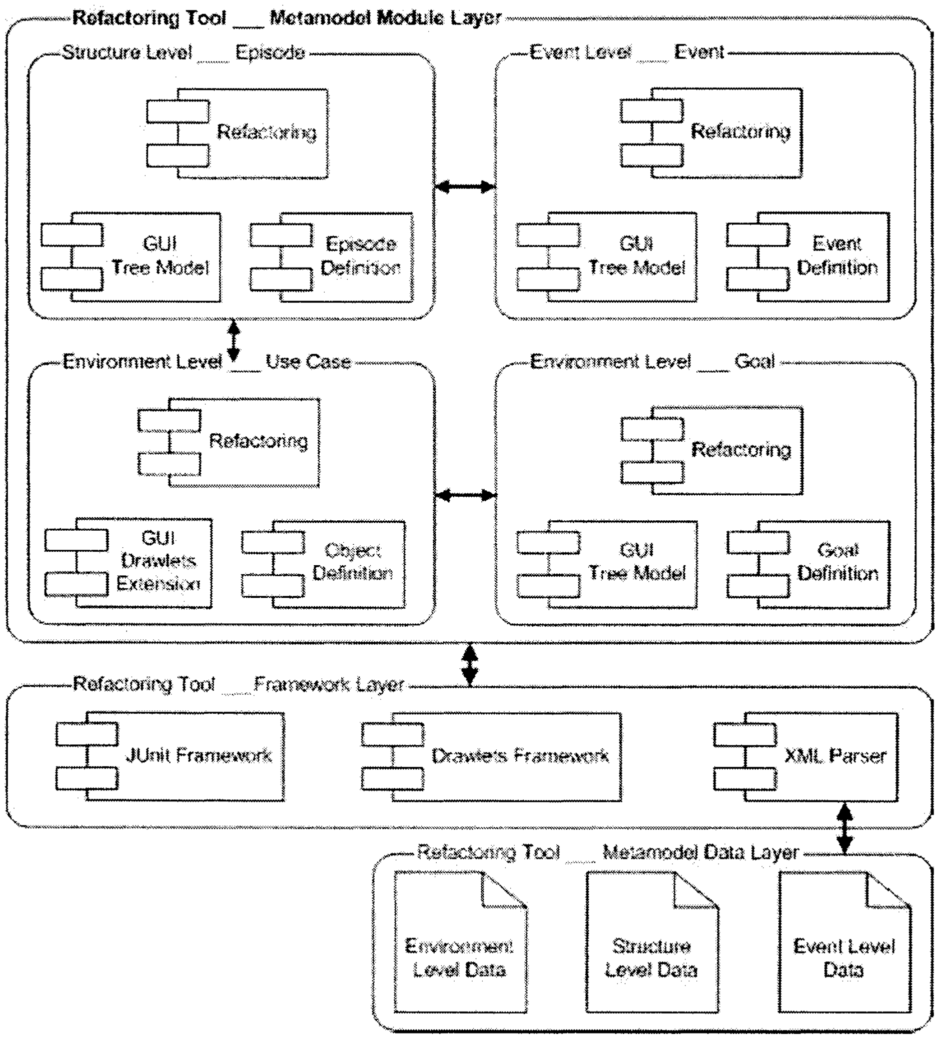


Figure 20: System Architecture of the Tool [XU04]

4.2.2 Refactoring Subsystem

The Refactoring subsystem is responsible for applying refactorings to the use case model. It checks preconditions of the refactoring automatically. The refactoring can only be applied when all the preconditions are met. Figure 21 shows the use case diagram for this subsystem. The *Model Designer* uses this subsystem to evolve the use case model by refactorings. These refactorings are divided into three levels: the environment level , the structure level and the event level.

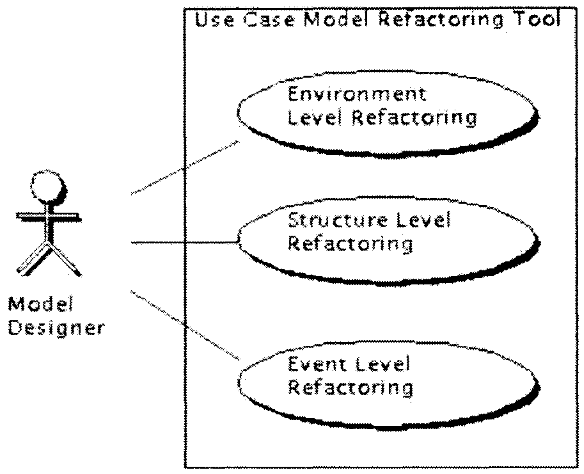

Use Case Model Refactoring Tool
Environment Level Refactoring
Structure Level Refactoring
Event Level Refactoring
Model Designer

Figure 21: Use Case Diagram of the Refactoring Subsystem

4.3 Data Structure and Storage

The use case model is stored in the XML format. A separate file is used to store the use case layer, episode layer and event layer respectively.

4.3.1 Environment Layer

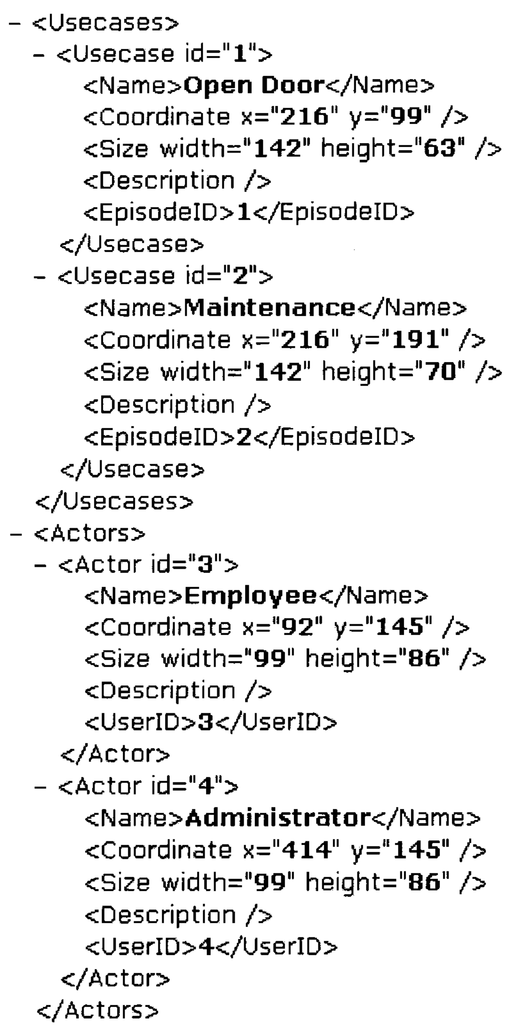
At the environment layer, there are five entities: *Model*, *Actor*, *Usecase*, *Relationship* and *Supplementary* entity. Figure 22 shows an example of XML representation of use case and actor. Figure 23 shows an example of XML representation of their relationships.

(1). A *Model* entity contains the model information and defines the possible sub-entity types in the model.

(2). An *Actor* entity contains the actor information: name, communicating use cases, parent actors, etc. The *id* attribute of an *Actor* entity is its identity. Other attributes: x, y, width and height, are used by the use case editor to store the position and size of the actor figure.

(3). A *Usecase* entity stores the use case information. Each *Usecase* entity has an identification attribute *id*. Other attributes are used by the use case editor, just like those in the *Actor* entity. An *EpisodeID* attribute stores the information of the referenced episode.

(4). A *Relationship* entity defines the relationship between two use cases, two actors or a use case and an actor. There are seven relationships defined at this level: *association*, *generalization*,



```
- <Usecases>
  - <Usecase id="1">
      <Name>Open Door</Name>
      <Coordinate x="216" y="99" />
      <Size width="142" height="63" />
      <Description />
      <EpisodeID>1</EpisodeID>
    </Usecase>
  - <Usecase id="2">
      <Name>Maintenance</Name>
      <Coordinate x="216" y="191" />
      <Size width="142" height="70" />
      <Description />
      <EpisodeID>2</EpisodeID>
    </Usecase>
  </Usecases>
- <Actors>
  - <Actor id="3">
      <Name>Employee</Name>
      <Coordinate x="92" y="145" />
      <Size width="99" height="86" />
      <Description />
      <UserID>3</UserID>
    </Actor>
  - <Actor id="4">
      <Name>Administrator</Name>
      <Coordinate x="414" y="145" />
      <Size width="99" height="86" />
      <Description />
      <UserID>4</UserID>
    </Actor>
  </Actors>
```

Figure 22: Example: XML Representation of Use Case and Actor

```
- <Relationship>
  - <Association>
    - <Actor-Usecase>
        <ActorRef>10</ActorRef>
        <UsecaseRef>12</UsecaseRef>
      </Actor-Usecase>
    - <Actor-Usecase>
        <ActorRef>10</ActorRef>
        <UsecaseRef>13</UsecaseRef>
      </Actor-Usecase>
    - <Actor-Usecase>
        <ActorRef>11</ActorRef>
        <UsecaseRef>13</UsecaseRef>
      </Actor-Usecase>
    </Association>
  - <Generalization>
    - <Actor-Actor>
        <ActorRef1>10</ActorRef1>
        <ActorRef2>14</ActorRef2>
      </Actor-Actor>
    - <Actor-Actor>
        <ActorRef1>11</ActorRef1>
        <ActorRef2>14</ActorRef2>
      </Actor-Actor>
    </Generalization>
  </Relationship>
```

Figure 23: Example: XML Representation of Relationship at the Environment Layer

inclusion, *extension*, *similarity*, *equivalence* and *precedence*. The *ActorRef* and *UsecaseRef* attributes refer to the *id* value of the corresponding actor and use case.

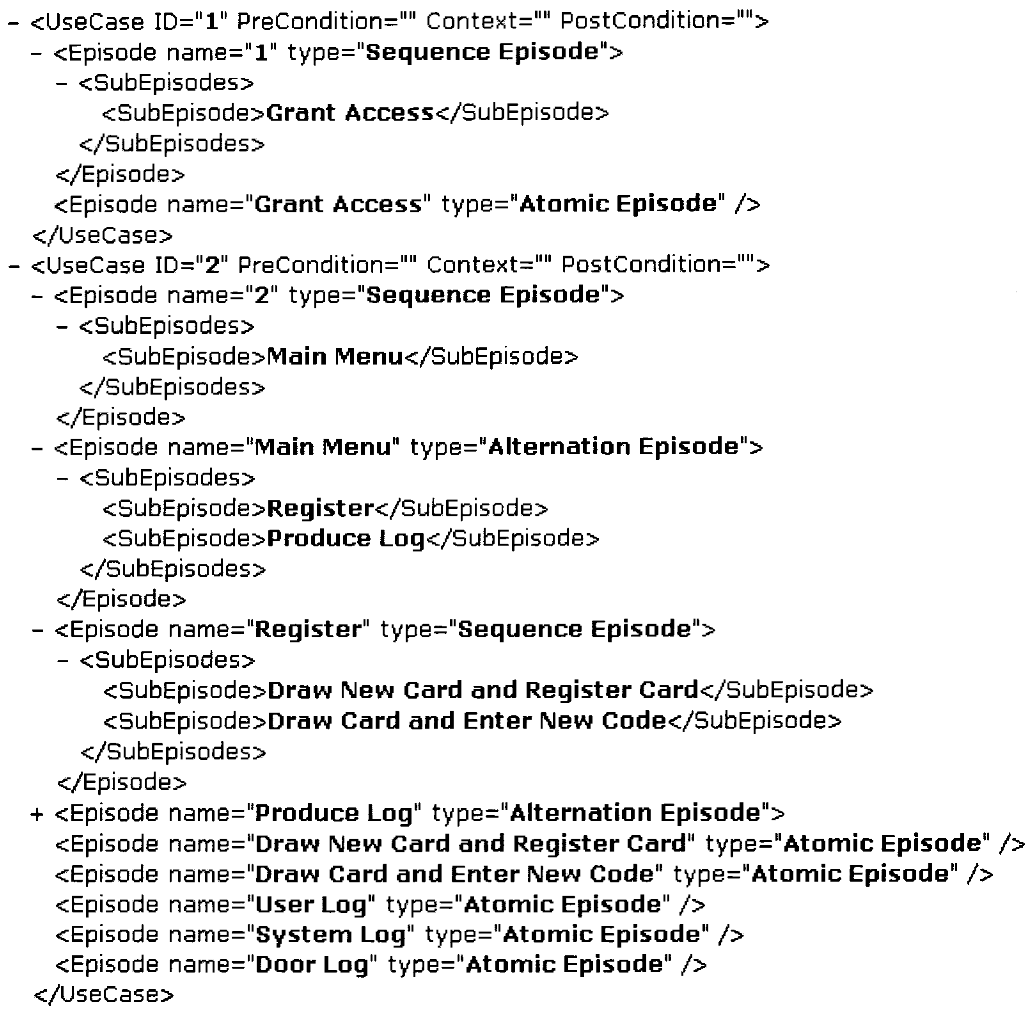
(5). The *Supplementary* entity is introduced from the UML semantics. It is divided into three categories: system bound, service bound and text label.

4.3.2 Structure Layer

At the structure level, *episode*, *subepisodes* and *subepisode* are defined. Figure 24 shows an example of XML representation at the structure layer.

(1). *Episode* is related to the use case. A use case can contain multiple episodes. There are three categories of episode: *primitive*, *composite* and *pseudo-episode*. The *primitive* episode can be shared among use cases and is unique in the use case model. There are four categories of *composite* episode: *sequence*, *parallel*, *alternation* and *iteration*.

(2). *Subepisodes* is used when *composite* episode is defined. It consists of *subepisode*.



```
- <UseCase ID="1" PreCondition="" Context="" PostCondition="">
  - <Episode name="1" type="Sequence Episode">
    - <SubEpisodes>
        <SubEpisode>Grant Access</SubEpisode>
      </SubEpisodes>
    </Episode>
    <Episode name="Grant Access" type="Atomic Episode" />
  </UseCase>
- <UseCase ID="2" PreCondition="" Context="" PostCondition="">
  - <Episode name="2" type="Sequence Episode">
    - <SubEpisodes>
        <SubEpisode>Main Menu</SubEpisode>
      </SubEpisodes>
    </Episode>
  - <Episode name="Main Menu" type="Alternation Episode">
    - <SubEpisodes>
        <SubEpisode>Register</SubEpisode>
        <SubEpisode>Produce Log</SubEpisode>
      </SubEpisodes>
    </Episode>
  - <Episode name="Register" type="Sequence Episode">
    - <SubEpisodes>
        <SubEpisode>Draw New Card and Register Card</SubEpisode>
        <SubEpisode>Draw Card and Enter New Code</SubEpisode>
      </SubEpisodes>
    </Episode>
  + <Episode name="Produce Log" type="Alternation Episode">
    <Episode name="Draw New Card and Register Card" type="Atomic Episode" />
    <Episode name="Draw Card and Enter New Code" type="Atomic Episode" />
    <Episode name="User Log" type="Atomic Episode" />
    <Episode name="System Log" type="Atomic Episode" />
    <Episode name="Door Log" type="Atomic Episode" />
  </UseCase>
```

Figure 24: Example: XML Representation at the Structure Layer

(3). *Subepisode* is included in the *subepisodes*, which can be *primitive* or *composite* episode.

4.3.3 Event Layer

At the event level, stimulus, action and response are defined as *PrimitiveEvent*. Figure 25 shows an example of XML representation at the event layer.

4.4 System Framework Design

The whole system can be divided into four major components: use case tool, goal tool, episode tool and event tool. This section describes each component in detail.

```
- <EventModel>
    <PrimitiveEvent Name="Display Message" Type="Response" />
    <PrimitiveEvent Name="Input Code" Type="Stimulus" />
    <PrimitiveEvent Name="Validate Code" Type="Action" />
  </EventModel>
```

Figure 25: Example: XML Representation at the Event Layer

4.4.1 Use Case Tool

Use Case Tool is designed for defining the use case model using use case, actor, user, goal and service. Figure 26 shows the class diagram of this component. The *frame* package is inherited from the Drawlets framework. The *object* package is used to define the entities and load/save data from/to XML files. The *refactoring* package implements use case refactorings. It also contains the unit testing program.

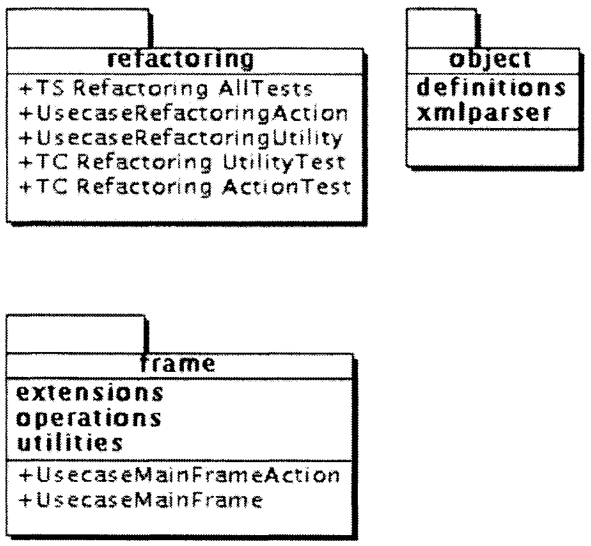


Figure 26: Class Diagram of Use Case Tool [XU04]

There are four major modules within *Use Case Tool*, which are *Drawlets Extension*, *Object Definition*, *Refactoring* and *XML Parser*. These modules are described in detail as follows.

Drawlets Extension

In *Drawlets Extension* module, the Drawlets framework is extended in the following ways.

(1). The *frame* is extended to support the MDI interface. The tool is implemented as an MDI application, which allows all the layers to appear in the same frame.

(2). The *frame* is extended to support loading and saving data. We also define our own toolbar and palette.

(3). The definition of *object* is extended in order to add new objects: use case, actor, service, etc. We also add some new modules for use case, episode, event, goal and task.

(4). *Figure* and *DrawingCanvas* are the most important interfaces in Drawlets. The *figure* interface is implemented for all entities like actor, use case, etc. so that these entities can be added to the *figure* container without any further operations. *DrawingCanvas* is implemented as the *figure* container in order to keep all the entities at runtime.

Object Definition

Object Definition module provides capabilities to define the use case model at the environment level, which includes use case, actor, relationship, service, etc. It also provides features to navigate from the use case to its structure level and from the actor to the goal model.

Figure 27 shows the class diagram of the *Object Definition* module. There are two parts in each object. One is the definition part, which records attributes of the object. The other is the operation part, which provides ways to create object, draw object and change object attributes. There are two types of objects: *Entity* where *EntityBase* and *EntityBaseTool* are the base classes, and *Relation* where *RelationBase* and *RelationBaseTool* are the base classes. These base classes are introduced further below.

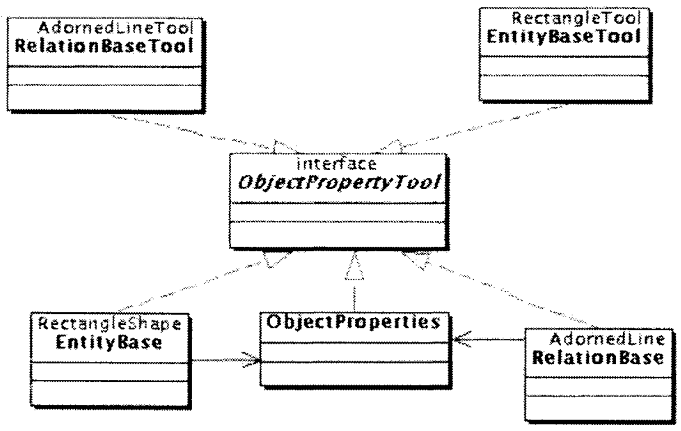


Figure 27: Class Diagram of Object Definition Module [XU04]

(a). EntityBase

EntityBase is the base class for the following objects: Actor, ServiceBound, SystemBound and UseCase. Figure 28 shows its class diagram. *EntityBase* is extended from *RectangleShape* in Drawlets. It defines a collection of entity properties. We implement the way to draw an entity and specify values for entity properties in its sub classes.

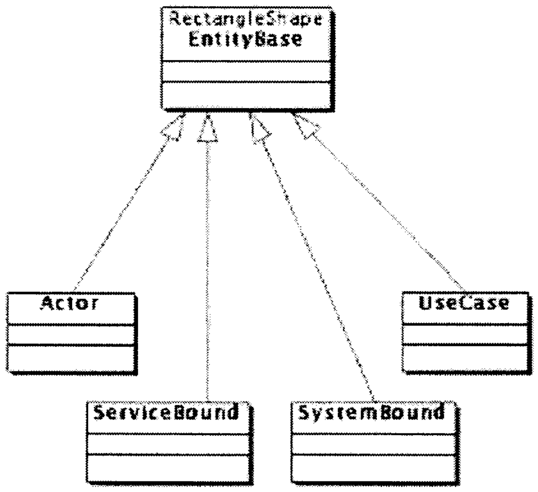


Figure 28: Class Diagram of EntityBase

(b). EntityBaseTool

EntityBaseTool is the base class for ActorTool, ServiceBoundTool, SystemBoundTool and UseCaseTool. Figure 29 shows its class diagram. *EntityBaseTool* is extended from *RectangleTool* in Drawlets. It defines how to access properties of an entity. In its sub classes, we define how to create a new entity and how to respond to external events either from mouse or from keyboard.

(c). RelationBase

RelationBase is the base class for relation objects: Generalization, Extension, Similarity, Equivalence, Association, Precedence and Inclusion. Figure 30 shows its class diagram. *RelationBase* is extended from *AdornedLine* in Drawlets. It defines relation properties and locations of a connection line between two entities. In its sub classes, we define how to draw the connection line with specified text.

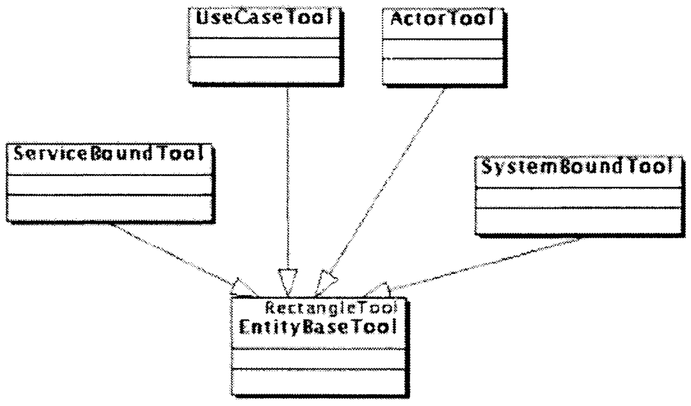


Figure 29: Class Diagram of EntityBaseTool

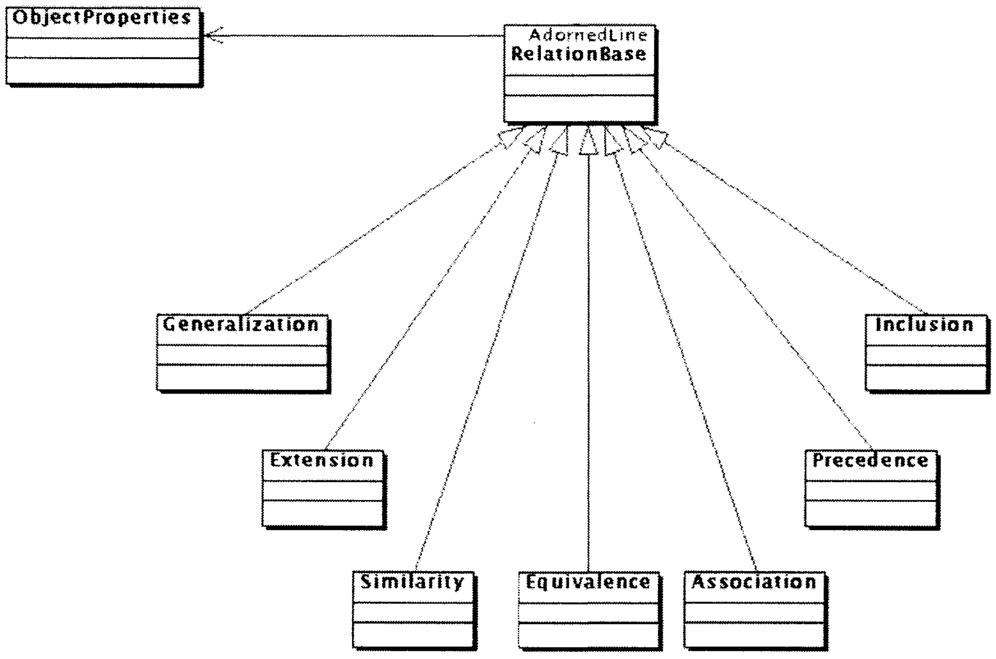


Figure 30: Class Diagram of RelationBase

(d). RelationBaseTool

RelationBaseTool is the base class for GeneralizationTool, ExtensionTool, SimilarityTool, EquivalenceTool, AssociationTool, PrecedenceTool and InclusionTool. Figure 31 shows its class diagram. *RelationBaseTool* is extended from *AdornedLineTool* in Drawlets. It defines how to access relation properties and how to check whether a relation can be created or not between two entities. For example, only *Association* relation is allowed between an *Actor* and a *UseCase*. Its sub classes specify how to create a relation between two selected entities.

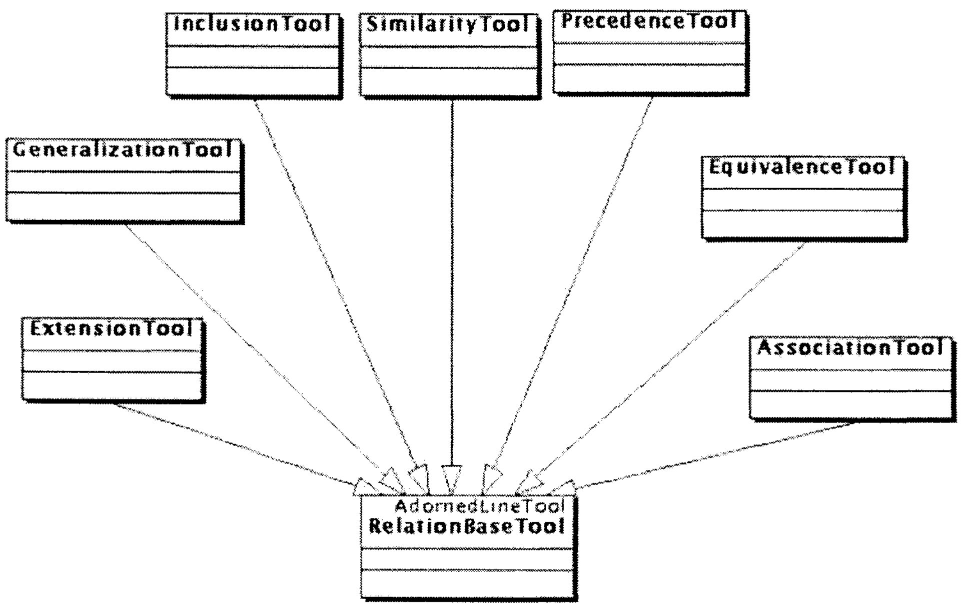


Figure 31: Class Diagram of RelationBaseTool

Refactoring

The *Refactoring* module implements a list of use case refactorings at the environment level. As shown in Table 3, there are thirty-eight refactorings implemented in this module. These refactorings are grouped into six categories: *Create an Environmental Entity, Delete an Environmental Entity, Change an Environmental Entity, Compose an Environmental Entity, Decompose an Environmental Entity, Move an Environmental Entity.*

The *Refactoring* module communicates with the structure level and event level to validate conditions of each refactoring. If conditions are met, it updates the use case model according to the refactoring specification in Appendix.

Table 3: Implemented Refactorings at the Environment Level

Category	Refactoring Name
Create an Environmental Entity	Create_Empty_Usecase Create_Empty_Actor Create_Empty_User Create_Empty_Service Create_Empty_Goal Create_Empty_Task
Delete an Environmental Entity	Delete_Unreferenced_Usecase Delete_Unreferenced_Actor Delete_Unreferenced_User Delete_Unreferenced_Service Delete_Unreferenced_Goal Delete_Unreferenced_Task
Change an Environmental Entity	Change_Usecase_Name Change_Actor_Name Change_User_Name Change_Service_Name Change_Goal_Name Change_Task_Name
Compose an Environmental Entity	Change_Parent_Usecase Change_Parent_Actor Usecase_Generalization_Merger Usecase_Inclusion_Merger Usecase_Extension_Merger Usecase_Precedence_Merger Usecase_Equivalence_Generation Reuse_Usecase Merge_Usecases Generalize_Usecases Merge_Actors Generalize_Actors
Decompose an Environmental Entity	Split_Usecase Usecase_Generalization_Generation Usecase_Inclusion_Generation Usecase_Extension_Generation Usecase_Precedence_Generation Split_Actor Actor_Generalization_Generation
Move an Environmental Entity	Move_Goal_To_Parent_Actor

Figure 32 shows the class diagram of the *Refactoring* module. Basically, use case refactorings are implemented in two classes: *UsecaseRefactoringAction* and *UsecaseRefactoringUtility*. *UsecaseRefactoringAction* defines conditions for use case refactorings at the environment level. *UsecaseRefactoringUtility* is used to apply refactorings to the use case model. It also provides interfaces to the structure level and event level so that the entity information at the environment level can be accessed.

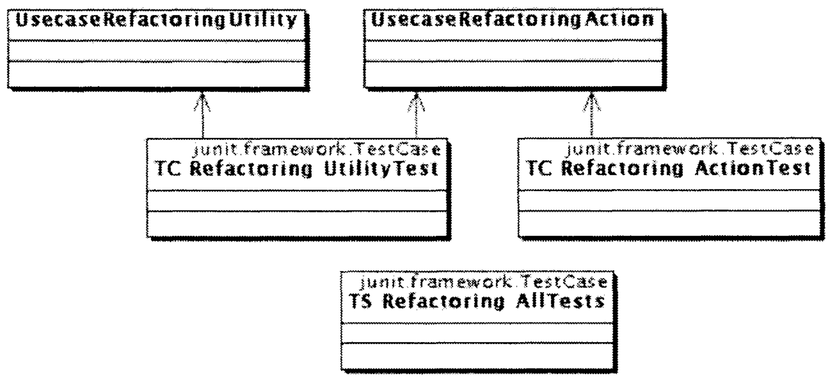


Figure 32: Class diagram of Refactoring Module [XU04]

Based on JUnit, *JUnit TestCases* is defined in the *Refactoring* module. It enables us to test refactorings automatically within the JUnit framework application. This is especially useful when the refactoring implementation needs to be changed frequently.

XML Parser

The *XML Parser* module is used to load/save information for the use case model from/to XML files. Figure 33 shows its class diagram. *XMLProcessor* parses the XML file and translates the XML structure into the use case model. It also calls related modules to create entities or relationships within the model. The XML DTD is defined so that the XML file can be validated by *XMLProcessor*. The *XMLGenerator* interface is implemented to save the DTD and the model information into the XML file.

4.4.2 Goal Tool

This component includes the following modules: *goal definition*, *data display* and *refactoring*. Figure 34 shows its class diagram. In the goal model, goals can be defined and saved through the graphical interface. Goals are related to actors. When refactorings are applied to actors, related goals are checked to validate if conditions are satisfied.

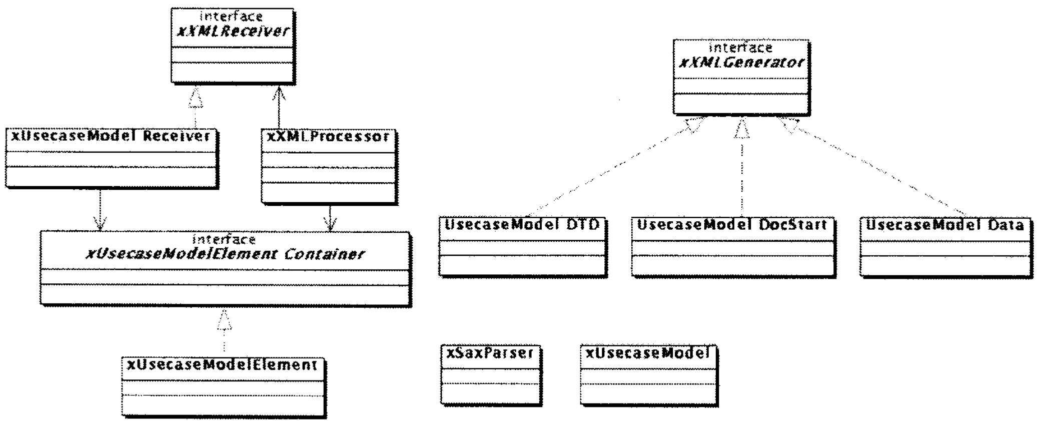


Figure 33: Class Diagram of XML Parser [XU04]

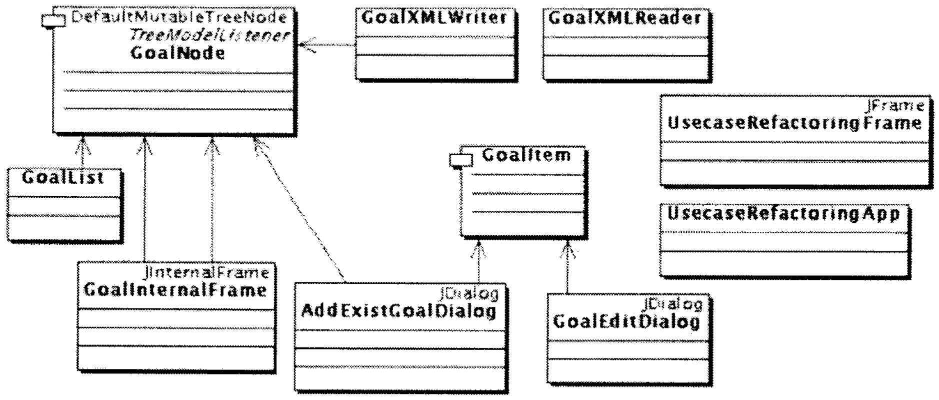


Figure 34: Class Diagram of Goal Module [XU04]

4.4.3 Episode Tool

This component includes the following modules: *episode definition*, *context/precondition/ postcondition definition*, *data displaying*, *data reading/writing* and *refactoring*. Figure 35 shows its class diagram. The episode model is defined in the *object* package. Episodes are displayed using tree and table structures. The *xml* package is used to read/write XML files. Refactorings at the structure level are defined in the *refactoringRule* package. The information for the episode model is shown in the frame, which is defined in the *frame* package. There are also some supporting packages: *dialog*, *popupMenu* and *action*.

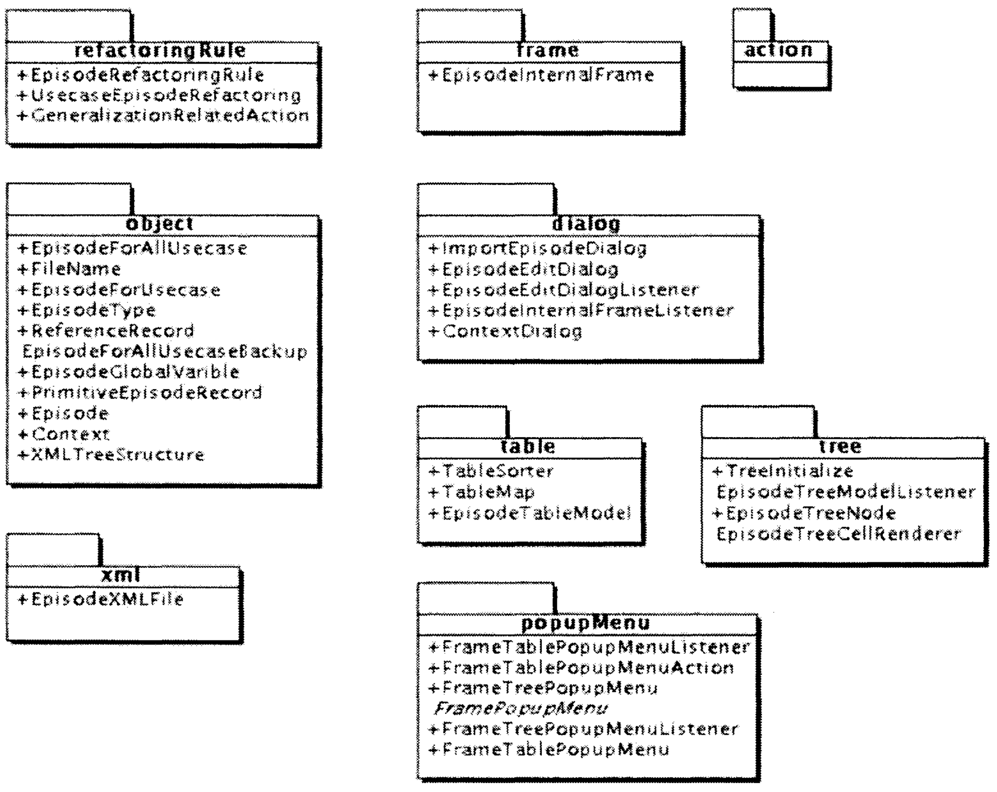


Figure 35: Class Diagram of Episode Tool [XU04]

This component is designed to define the structure of the use case in detail. In the episode model, users can define context, precondition and postcondition of use case. They can add primitive and composite episodes. Episodes can be shared among different episode models.

The feature is provided to navigate to other levels from the structure level within the use case model. The only way to enter into the episode layer is to select a use case in the use case layer. Similarly, the event layer can be accessed by selecting an episode in the episode layer. The episode layer is synchronized with the use case layer and event layer.

The *Refactoring* module within this component implements twelve use case refactorings at the structure level. As shown in Table 4, these refactorings are grouped into seven categories: *Create a Structural Entity*, *Delete a Structural Entity*, *Change a Structural Entity*, *Compose a Structural Entity*, *Decompose a Structural Entity*, *Move a Structural Entity*, and *Convert a Structural Entity*.

Table 4: Implemented Refactorings at the Structure Level

Category	**Refactoring Name**
Create a Structural Entity	Create_Empty_Episode
Delete a Structural Entity	Delete_Unreferenced_Episode
Change a Structural Entity	Change_Episode_Name
Compose a Structural Entity	Episode_Sequence_Merger Episode_Alternation_Merger Episode_Parallel_Merger Generalize_Episodes
Decompose a Structural Entity	Episode_Sequence_Generation Episode_Alternation_Generation Episode_Parallel_Generation
Move a Structural Entity	Move_Episode_To_Parent_Usecase
Convert a Structural Entity	Convert_Episode_To_Usecase

4.4.4 Event Tool

This component includes the following modules: *event definition*, *data displaying*, *data reading/writing* and *refactoring*. Figure 36 shows its class diagram. Similar to the episode layer, the *object* package is used to define the event model. It also uses tree and table structures to display events and the event model. The *xml* package is used for reading/writing XML files. Refactorings at the event level are defined in the *refactoringRule* package. Information for the event model is shown in the frame, which is defined in the *frame* package. There are also some supporting packages: *dialog*, *popupMenu* and *action*.

This component supports the synchronization between the event layer and other layers. Events are shared by all the episodes in the use case model. This component provides the capability to navigate from the event level to the structure level, and then to the environment level within the use case model.

The *Refactoring* module within this component implements three use case refactorings at the event level. As shown in Table 5, these refactorings are grouped into three categories: *Create an Event*, *Delete an Event*, and *Change an Event*.

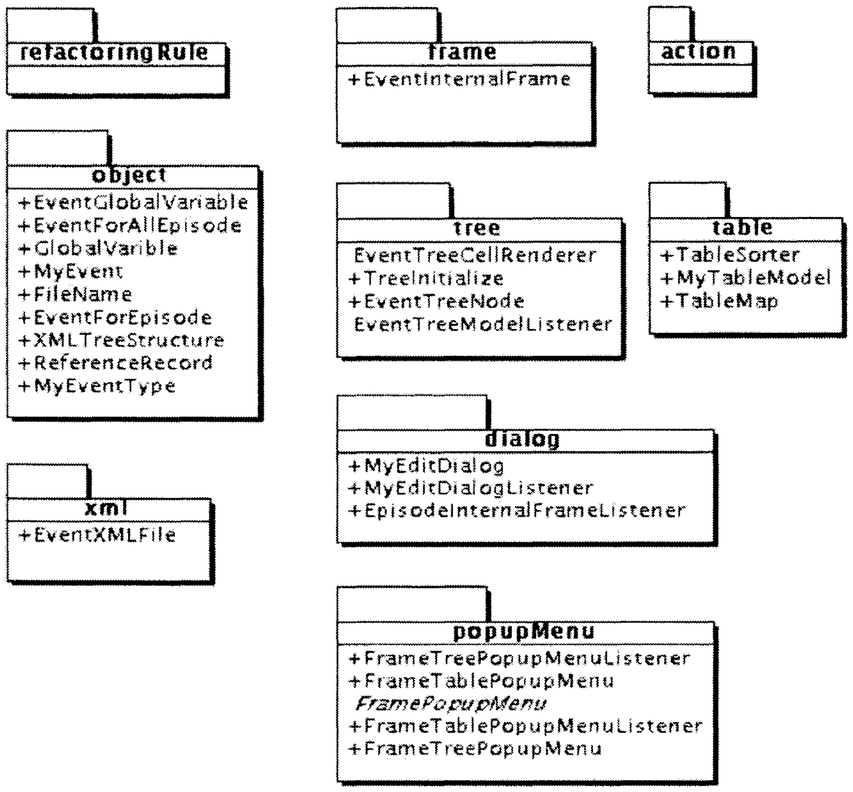


Figure 36: Class Diagram of Event Module [XU04]

Table 5: Implemented Refactorings at the Event Level

Category	Refactoring Name
Create an Event	Create_Unreferenced_Event
Delete an Event	Delete_Unreferenced_Event
Change an Event	Change_Event_Name

4.5 Summary

The use case modeling and refactoring tool is implemented based on the use case metamodel. As a modeling tool, it brings many benefits to the use case model.

- Understandability: The use case model can be specified with much more details than the one in UML. It helps users to understand the system requirement better. While the environment level gives an overview of the requirement, the structure level illustrates the structure of each use case. The event level provides more detailed information. Therefore, a use case model can be built with the tool to specify system requirements in detail. These requirements are organized in an easily understandable way.
- Reusability: Once the episode and event are defined, they can be reused within the use case model. This prevents redundant definition for the same episode or event. It helps reuse existing work.
- Changeability: The tool facilitates modifications to the use case model by providing supporting features.
- Traceability: The tool provides a mechanism to trace related information across different levels in the use case model conveniently. For example, for a given episode, we can get all of its events, and related use cases that it is involved.

As a use case refactoring tool, it implements fifty-three use case refactorings. These refactorings are at different levels and are divided into different categories. With these refactorings, we can restructure the use case model to further improve its understandability, reusability, changeability and traceability. The tool ensures that conditions for the refactoring are met before changing the model. With the tool support, use case models can be refactored efficiently and automatically. The tool avoids errors inherent in the manual process of restructuring the use case model. After applying use case refactorings, the tool ensures that models at different levels are synchronized.

Chapter 5

Case Studies

After the refactoring tool is built, we carry out some case studies to evaluate the tool, validate use case refactorings and understand the whole refactoring process. These case studies are initially conducted with two master students: Jian Xu mainly works on the Automatic Teller Machine (ATM) [XU04] while Wei Yu focuses on the Video Store System (VSS)[YU04]. However, I carry out these case studies again with revised steps. This chapter introduces these case studies with some new perspectives. To put these case studies in the context, the thesis introduces some background information. It explains how these case studies are designed and what are the experimental results. In Section 5.1, it describes how the structure of a relatively more complex use case model is improved by refactoring step by step. In Section 5.2, it introduces how the use case model is created incrementally with use case refactorings starting from a simple model. The thesis presents evaluation results of these case studies in Section 5.3. For the sake of conciseness, some episode models are omitted in the ATM case study within Section 5.2.

5.1 Video Store System (VSS)

In this case study, we build a relatively complex use case model based on requirements of the Video Store System. Then we improve the structure of this model using refactorings. This section illustrates the whole process.

5.1.1 System Description

The Video Store System (VSS) is a computer-based business management system. The purpose of the VSS is to facilitate the staff to manage renting movies, video games and related equipments for a small chain of stores scattered throughout the city. The system manages the inventory of rental items, keeps track of all transactions, rental history and status of each customer member.

Basic Requirements

The system provides the following features:

- **Inventory Management**: The inventory is updated upon purchasing and receiving videotapes, games or equipments. The inventory is maintained by the store manager or someone authorized. Each store maintains its own inventory. The inventory can be searched, but cannot be updated by other stores.
- **Customer Maintenance**: The detailed information on customers is maintained by the store manager or someone authorized. The customer is assigned a membership card and a password for web access once the prerequisite is met. The VSS system also issues the dependent membership for the child. Dependent members must be associated with a primary member. All stores share a common list of customer members so that the customer membership is valid in all stores once established.
- **Rental Service**: Customers may rent videotapes, video games, and equipments through a store staff. The system verifies the customer membership, checks conditions, and records transaction details. Various rental conditions are applied to a child dependent member on what and how long the child can rent.
- **Reservation Service**: The VSS provides the reservation service. Customers may reserve videotapes, video games, and equipments for a specific date. Staff receives an reminder of such reservation on the screen when the item becomes available.
- **Inquiry Service**: The staff is able to search for movie titles on behalf of customers. If the item is not found in the store, it is possible to search other stores.
- **Web Service**: The VSS system provides the web access service. Customers can login at the website of the video store using their own membership card and password. Then they can search and reserve items including videotapes, games and equipments.

Actor Characteristics

The primary actor of the VSS is the video store staff. Customers of the video store access the VSS web service module through the internet. There are four actors in the system as follows:

- **Manager Staff**: manager or owner of the video store. Manager staff can use all features provided by the VSS, including the inventory management, customer maintenance, video rentals, video reservation, video return, video inquiries and promotion service.

- **Staff**: a regular employee, who is the primary actor of the system. Staff members use the majority of the functionality provided by the VSS, including video rentals, video reservation, video return and video inquiries.

- **Customer**: anyone who is registered as a primary member of the VSS. Customers interact with the system directly only when they perform web-reservation or web-search. The *Staff* member or *Manager Staff* performs all the other services of the VSS on behalf of customers.

- **Dependent Customer**: any child under 18 years old who is registered as a dependent member of a primary member. Dependent customers interact with the system directly only when they perform the web-reservation or web-search. The dependent customer depends on the corresponding primary customer. They use the same account for payment. Rental items are accumulated within the primary member and his/her dependent member. The *Staff* member or *Manager Staff* actor performs all the other services of the VSS on behalf of dependent customers.

5.1.2 VSS Initial Use Case Model

Based on requirements described above, we define actors, use cases, and use case relationships. Figure 37 shows the use case model at the environment level. It defines four actors as described in Section 5.1.1: Staff, Manager Staff, Customer and Dependent Customer. There are twelve use cases within this use case model.

Figure 37 only gives an overview of the use case model. The detailed system behavior is not illustrated. To show the internal structure of each use case, a list of episode models are presented as follows using episode trees.

Episode model for the use case "Video Rental"

```
(sequence)
 1 Input the membership information
 2 Input the video item information
 3 (alternation) Check if the video item is available
   3.1 Yes: register the new video loan
   3.2 No: show the unavailable message
```

Episode model for the use case "Video Return"

```
(sequence)
 1 Input the membership information
 2 Input the video item information
 3 Remove the corresponding loan on this video item
 4 Calculate the rental fee
 5 (alternation) Check if there is a reservation on this item
   5.1 Yes: send the reminder message to the customer reserving this item
   5.2 No: print out the receipt for the rental fee
```

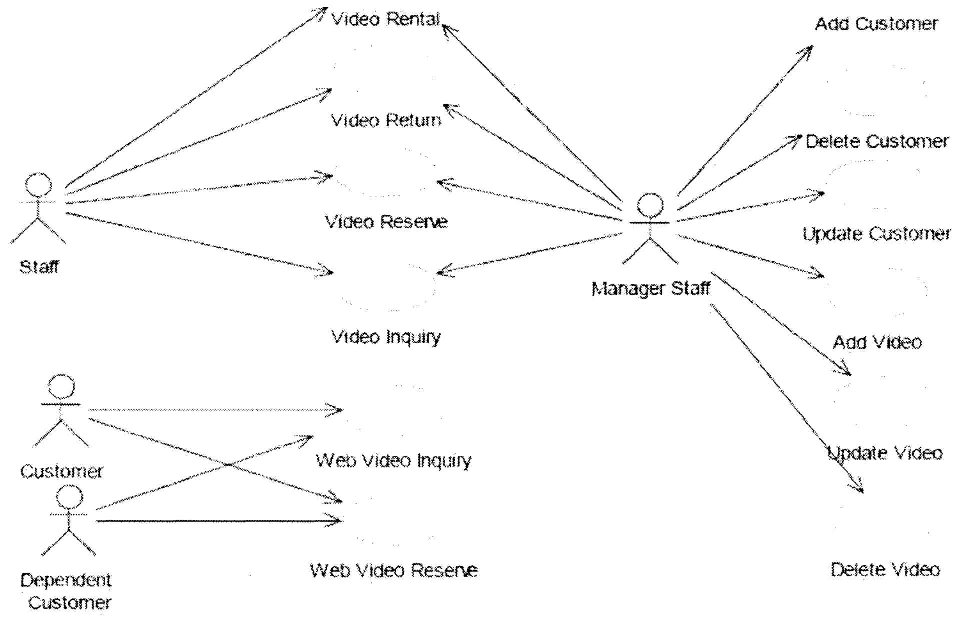


Figure 37: VSS Initial Use Case Model [YU04]

Episode model for the use case "Video Reserve"

(sequence)

1 Input the membership information
2 Identify the video item information
3 (alternation) Check if the reservation condition on this item is satisfied
  3.1 Yes: place a new reservation on this item
  3.2 No: show the unavailable message

Episode model for the use case "Video Inquiry"

(sequence)

1 (user) Input the search criteria
2 Search the video item
3 Return the search result

Episode model for the use case "Add Customer"

(sequence)

1 Input the registration information for the customer
2 Search for the customer
3 (alternation) Check if the customer already exists in the database
  3.1 Yes: return the existing message
  3.2 No: (sequence) issue the card for the customer
    3.2.1 Add the customer with corresponding information to the system
    3.2.2 Generate the membership card to the customer

```
              3.2.3 Return the successful message
```

Episode model for the use case "Update Customer"

```
(sequence)
  1 Input the membership information
  2 Search for the customer
  3 (alternation) Check if the customer already exists in the database
    3.1 Yes: (sequence) update customer's information
              3.1.1 Input the necessary information for the customer
              3.1.2 Update the database
              3.1.3 Return the successful message
    3.2 No: return the error message
```

Episode model for the use case "Delete Customer"

```
(sequence)
  1 Input the membership information
  2 Search for the customer
  3 (alternation) Check if the customer already exists in the database
    3.1 Yes: (sequence) delete the customer
              3.1.1 Delete the customer from the database
              3.1.2 Return the successful message
    3.2 No: return the error message
```

Episode model for the use case "Add Video"

```
(sequence)
  1 Input the video item information
  2 Search the video item
  3 (alternation) Check if the video item already exists
    3.1 Yes: return the message that the video item already exists
    3.2 No: (sequence) add the video item
              3.2.1 Add video item to the database
              3.2.2 Return the successful message
```

Episode model for the use case "Update Video"

```
(sequence)
  1 Input the video item information
  2 Search the video item
  3 (alternation) Check if the video item already exists
    3.1 Yes: (sequence) update the related information on this video item
              3.1.1 Input the information for the video item
              3.1.2 Update the database
              3.1.3 Return the successful message
    3.2 No: return the unavailable message
```

Episode model for the use case "Delete Video"

```
(sequence)
  1 Input the video item information
  2 Search the video item
```

```
3 (alternation) Check if the video item already exists
  3.1 Yes: (sequence) delete the video item
          3.1.1 Delete the item from the database
          3.1.2 Return the successful message
  3.2 No: return the unavailable message
```

Episode model for the use case "Web Video Inquiry"

```
(sequence)
  1 (user) Login to the website
  2 (user) Input the search criteria
  3 Search the video item
  4 Return the search result
```

Episode model for the use case "Web Video Reserve"

```
(sequence)
  1 (user) Login to the website
  2 Input the video title
  3 (alternation) Check if the reservation condition on this item is satisfied
    3.1 Yes: place a new reservation on this item
    3.2 No: show the unavailable message
```

5.1.3 Refactoring the VSS Use Case Model

Based on the internal structure of each use case defined in Section 5.1.2, it is apparent that it is necessary to restructure the use case model to improve its understandability, reusability and maintainability. This section illustrates detailed refactoring steps to achieve these improvements and motivations behind of each step.

Step 1: Use the *Change_Parent_Actor* refactoring to establish generalization relationships between actor "Staff" and "Manager Staff", "Customer" and "Dependent Customer".

From the use case model shown in Figure 37, we can see that the actor "Manager Staff" participates in all use cases that the actor "Staff" participates in. There is an inherent generalization relationship between these two actor. Therefore, the *Change_Parent_Actor* refactoring is used to set the actor "Staff" as the parent of the actor "Manager Staff". Similarly, the actor "Dependent Customer" participates in the same set of use cases that the actor "Customer" participates in. From actor characteristics defined in Section 5.1.1, it will be more nature to set the actor "Customer" as the parent of the actor "Dependent Customer". This is also achieved by applying the *Change_Parent_Actor* refactoring.

Figure 38 shows the VSS use case model after these refactorings. The actor "Manager Staff" inherits association relationships to use cases "Video Rental", "Video Return", "Video Inquiry" and "Video Reserve" from its parent actor "Staff". Similarly, the actor "Dependent Customer" inherits association relationships to use cases "Web Video Inquiry" and "Web Video Reserve" from its parent actor "Customer".

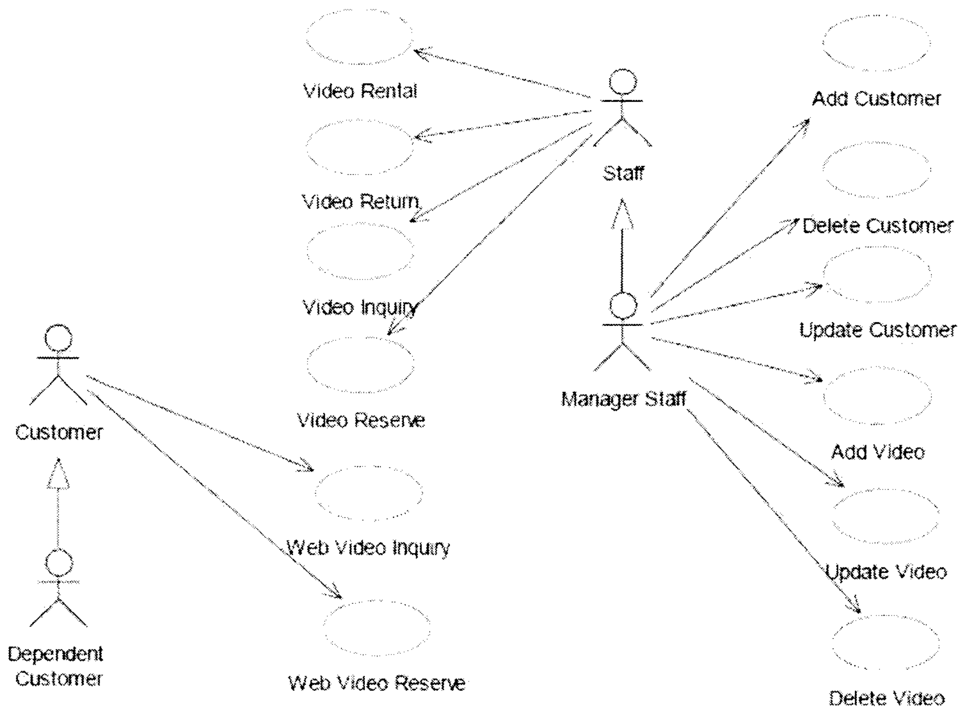


Figure 38: VSS Case Study: After Step 1 [YU04]

Step 2: Use the *Usecase_Inclusion_Generation* refactoring to break up the use case "Web Video Inquiry" into the new use case "Web Video Inquiry" and the new use case "Video Search". The new use case "Web Video Inquiry" includes the use case "Video Search". Then the *Usecase_Equivalence_Generation* refactoring is applied to make the use case "Video Search" equivalent to the use case "Video Inquiry".

In Section 5.1.2, we present episode models for use cases "Web Video Inquiry" and "Video Inquiry". For the purpose of convenience, they are listed here again.

Episode model for the use case "Web Video Inquiry"

(sequence)
1 (user) Login to the website
2 (user) Input the search criteria
3 Search the video item
4 Return the search result

Episode model for the use case "Video Inquiry"

(sequence)
  1 (user) Input the search criteria
  2 Search the video item
  3 Return the search result

As shown above, the episode tree of the use case "Video Inquiry" is a subtree of the episode tree of the use case "Web Video Inquiry". We can use refactorings to reduce such redundancy and improve the reusability.

Firstly, We use the *Usecase_Inclusion_Generation* refactoring to break up the use case "Web Video Inquiry" into two use cases as shown in Figure 39.

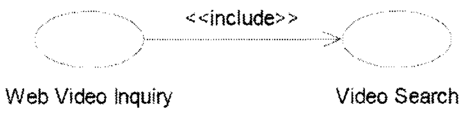


Figure 39: VSS Case Study: Step 2 - *Usecase_Inclusion_Generation* Refactoring [YU04]

After applying the *Usecase_Inclusion_Generation* refactoring, a new use case "Video Search" is created. Its episode tree is the same as the one for the use case "Video Inquiry". The new use case "Web Video Inquiry" includes the use case "Video Search". The inclusion point "video search" is added into the use case "Web Video Inquiry". The new use case "Web Video Inquiry" includes the use case "Video Search" at this point. Since it includes the episode tree of the use case "Video Search", the corresponding episode subtree is removed from the new use case "Web Video Inquiry" to avoid the redundancy. The episode model for the use case "Web Video Inquiry" after the *Usecase_Inclusion_Generation* refactoring is shown as follows.

Revised Episode Model for the Use Case "Web Video Inquiry"

(sequence)
  1 (user) Login to the website
  2 Video search (*inclusion*)

Secondly, We apply the *Usecase_Equivalence_Generation* refactoring to use cases "Video Search" and "Video Inquiry" because they have the equivalent episode model. Figure 40 shows their relationship after applying the *Usecase_Equivalence_Generation* refactoring.

Step 3: Use the *Reuse_Usecase* refactoring to remove the use case "Video Search".

As shown in Step 2, use cases "Video Inquiry" and "Video Search" are equivalent. To reduce the redundancy further, we can remove one of them from the use case model without changing the

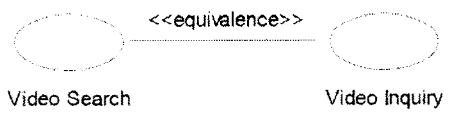


Figure 40: VSS Case Study: After Step 2

behavior of the use case model. However, we cannot simply remove one of these two use cases directly because the use case "Video Inquiry" is used by the actor "Staff" while the use case "Video Search" is included by the use case "Web Video Inquiry". We have to restructure the use case model first before removing one of them safely. This can be achieved by applying the *Reuse_Usecase* refactoring. After this refactoring, the use case "Web Video Inquiry" includes the use case "Video Inquiry" instead of "Video Search". The representation of the episode model in the use case "Web Video Inquiry" is changed. The inclusion point for this inclusion relationship refers to the use case "Video Inquiry". Its episode model after the refactoring is as follows:

Episode Model for the Use Case "Web Video Inquiry" after the Reuse_Usecase Refactoring

(sequence)
  1 (user) Login to the website
  2 Video inquiry (*inclusion*)

After applying the "Reuse_Usecase" refactoring, the use case "Video Search" is deleted from the use case model.

Step 4: Use the *Generalize_Usecases* refactoring to create the use case "Video Maintenance".

Use cases "Add Video", "Delete Video" and "Update Video" are related to the video maintenance. From episode models presented in Section 5.1.2, we can see that they share some common episodes. Their episode tree structures are similar. It is preferable to restructure these use cases to remove redundant episodes and illustrate the nature that they belong to the same family. We can apply the *Generalize_Usecases* refactoring to achieve this improvement. Figure 41 shows the result after refactoring these use cases.

The *Generalize_Usecases* refactoring creates a new use case "Video Maintenance". Common episodes within use cases "Add Video", "Delete Video" and "Update Video" are moved to this use case. Its episode model is as follows:

Episode model for use case "Video Maintenance"

(sequence)
  1 Input the video item information

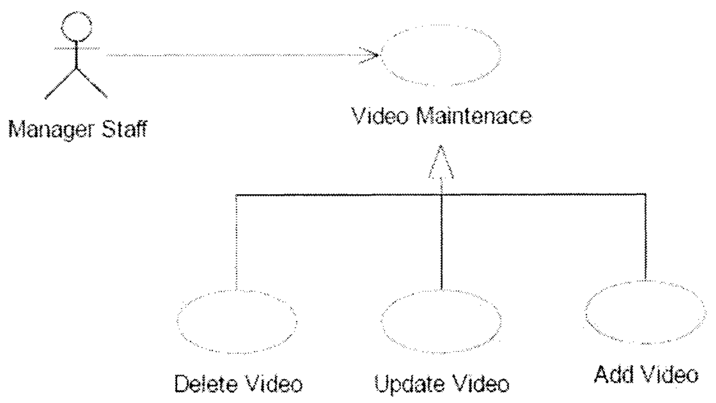


Figure 41: VSS Case Study: After Step 4 [YU04]

```
  2 Search the video item
  3 Check if the video item already exists
```

After applying the *Generalize_Usecases* refactoring, the use case "Video Maintenance" takes over the relationship to the actor "Manager Staff" from its child use cases. Use cases "Add Video", "Delete Video" and "Update Video" inherit the relationship with this actor and episodes in use case "Video Maintenance". New episode models for these use cases are as follows:

Episode model for the use case "Add Video"

```
(sequence)
  1 Input the video item information (inherited)
  2 Search the video item (inherited)
  3 (alternation) Check if the video item already exists (inherited)
    3.1 Yes: return the message that the video item already exists
    3.2 No: (sequence) add the video item
              3.2.1 Add video item to the database
              3.2.2 Return the successful message
```

Episode model for the use case "Update Video"

```
(sequence)
  1 Input the video item information (inherited)
  2 Search the video item (inherited)
  3 (alternation) Check if the video item already exists (inherited)
    3.1 Yes: (sequence) update the related information on this video item
              3.1.1 Input the information for the video item
              3.1.2 Update the database
              3.1.3 Return the successful message
    3.2 No: return the unavailable message
```

Episode model for the use case "Delete Video"

```
(sequence)
  1 Input the video item information (inherited)
  2 Search the video item (inherited)
  3 (alternation) Check if the video item already exists (inherited)
    3.1 Yes: (sequence) delete the video item
             3.1.1 Delete the item from the database
             3.1.2 Return the successful message
    3.2 No: return the unavailable message
```

Step 5: Use the *Generalize_Usecases* refactoring to create the use case "Customer Maintenance".

Similar to the Step 4, use cases "Add Customer", "Delete Customer" and "Update Customer" are related to the customer maintenance. They share some common episodes and their episode tree structures are similar. We can apply the *Generalize_Usecases* refactoring to group them into one family. Figure 42 shows the result after refactoring these use cases.

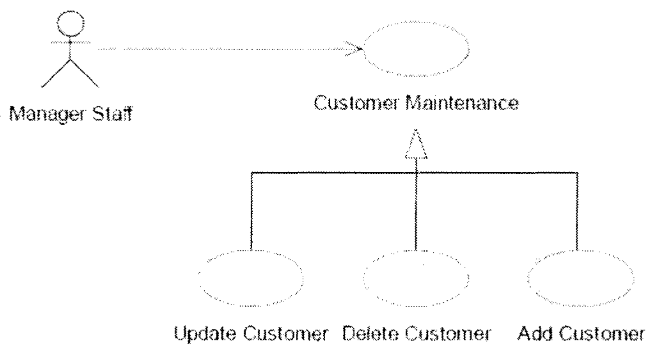


Figure 42: VSS Case Study: After Step 5 [YU04]

After applying the *Generalize_Usecases* refactoring, a new use case "Customer Maintenance" is created. Common episodes within use cases "Add Customer", "Delete Customer" and "Update Customer" are moved to this use case. The use case "Customer Maintenance" takes over the relationship to the actor "Manager Staff" from its child use cases. Use cases "Add Video", "Delete Video" and "Update Video" inherit the relationship with this actor and episodes in the use case "Customer Maintenance". Episode models for these use cases are as follows:

Episode model for the use case "Customer Maintenance"

```
(sequence)
  1 Search for the customer
  2 Check if the customer already exists in the database
```

Episode model for the use case "Add Customer"

```
(sequence)
  1 Input the registration information for the customer
  2 Search for the customer (inherited)
  3 (alternation) Check if the customer already exists in the database (inherited)
    3.1 Yes: return the existing message
    3.2 No: (sequence) issue the card for the customer
              3.2.1 Add the customer with corresponding information to the system
              3.2.2 Generate the membership card to the customer
              3.2.3 Return the successful message
```

Episode model for the use case "Update Customer"

```
(sequence)
  1 Input the membership information
  2 Search for the customer (inherited)
  3 (alternation) Check if the customer already exists in the database (inherited)
    3.1 Yes: (sequence) update customer's information
              3.1.1 Input the necessary information for the customer
              3.1.2 Update the database
              3.1.3 Return the successful message
    3.2 No: return the error message
```

Episode model for the use case "Delete Customer"

```
(sequence)
  1 Input the membership information
  2 Search for the customer (inherited)
  3 (alternation) Check if the customer already exists in the database (inherited)
    3.1 Yes: (sequence) delete the customer
              3.1.1 Delete the customer from the database
              3.1.2 Return the successful message
    3.2 No: return the error message
```

After all these refactorings, we achieve a new use case model as shown in Figure 43. This model is evolved from the initial model as shown in Figure 37. Compared with the initial model, the final model reduces the redundancy by making the use case "Web Video Inquiry" to include "Video Inquiry", and creating parent use cases "Video Maintenance" and "Customer Maintenance". It illustrates requirements better by grouping use cases "Add Video", "Delete Video", "Update Video" into the video maintenance category and grouping use cases "Add Customer", "Delete Customer", "Update Customer" into the customer maintenance category.

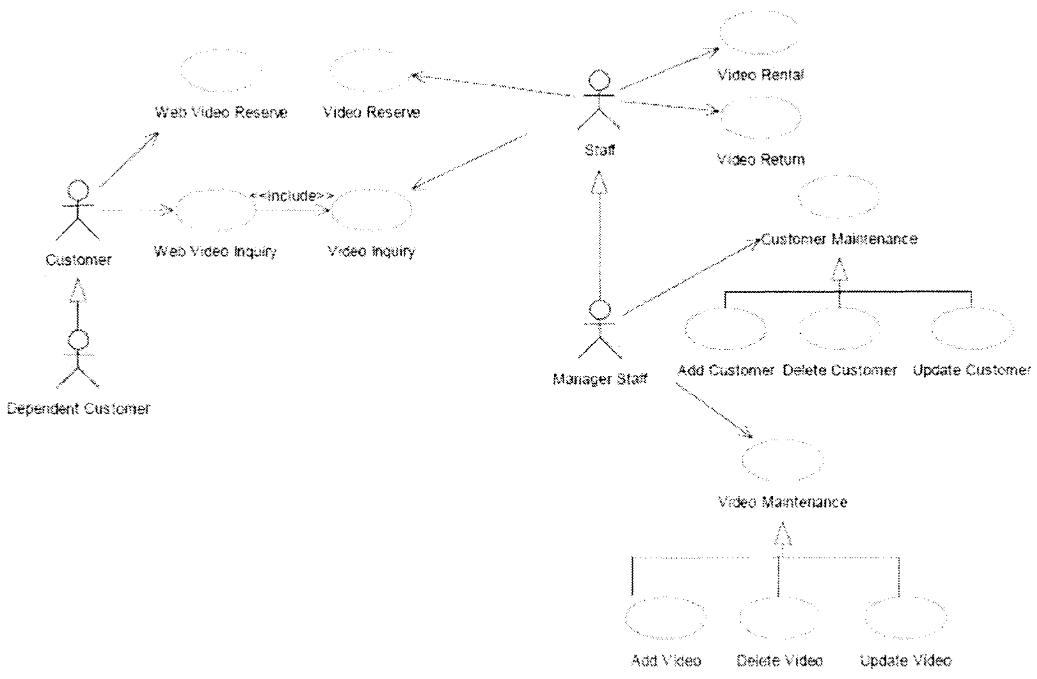


Figure 43: VSS Final Use Case model [YU04]

5.2 Automatic Teller Machine (ATM)

This section illustrates how we build the use case model incrementally with use case refactorings and extensions. We start from a simple model. Based on additional requirements, we improve the internal structure of the use case model with use case refactorings, then we extend the model to include these additional requirements.

5.2.1 Requirements of the ATM System

The Automatic Teller Machine (ATM) system is a classic example that is widely used in lots of systems for testing and validating functionalities. It is investigated thoroughly and referenced very often. The following description of the requirements for the ATM system is based on Professor Russell C. Bjorks ATM Simulation example used in his course [BJO].

The ATM system provides the ability for banking customers to deposit, withdraw, and transfer funds. It also allows them to inquire about account balances. The ATM system supports a network including both human cashiers and Automatic Teller Machines (ATMs) to be shared by a consortium of banks. Each bank provides its own computers to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter the account and transaction data. Automatic teller machines communicate with a central computer which clears transactions with appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires record keeping and security provisions, the system must handle concurrent accesses. The banks will provide their own software for their own computers.

The ATM is used by customers of a bank. Each customer has two accounts: a checking account and a savings account. Each customer has a customer number and a Personal Identification Number (PIN). Both must be typed to gain access to the accounts. Once they have gained access, the customer can select an account (checking or savings). The balance of the selected account is displayed (initially zero). Then the customer can deposit and withdraw money. The application terminates when the user selects *exit* rather than an account.

The ATM serves one customer at a time. A customer will be required to insert an ATM card and enter a Personal Identification Number (PIN) — both of which will be sent to the bank for validation as part of each transaction. The customer will then be able to perform one or more transactions. The card will be retained in the machine until the customer indicates that he/she desires no further transactions, at which point it will be returned — except as noted below.

The ATM must be able to provide the following services to the customer:

1. A customer must be able to make a cash withdrawal from any suitable account linked to the card, in multiples of \$20.00. The approval must be obtained from the bank before cash is dispensed.

2. A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or checks in an envelope. The customer will enter the amount of the deposit into the ATM, subject to the manual verification when the envelope is removed from the machine by an operator. Approval must be obtained from the bank before physically accepting the envelope.

3. A customer must be able to make a transfer of money between any two accounts linked to the card.

4. A customer must be able to make a balance inquiry of any account linked to the card.

5. A customer must be able to abort a transaction in progress by pressing the Cancel key instead of responding to a request from the machine.

The ATM communicates each transaction to the bank and obtains verification that it is allowed by the bank. Ordinarily, a transaction will be considered complete by the bank once it has been approved. In the case of a deposit, a second message will be sent to the bank indicating that the customer has deposited the envelope. (If the customer fails to deposit the envelope within the timeout period, or presses cancel instead, no second message will be sent to the bank and the deposit will not be credited to the customer.)

If the bank determines that the customer's PIN is invalid, the customer will be required to re-enter the PIN before a transaction can proceed. If the customer is unable to successfully enter the PIN after three attempts, the card will be permanently retained by the machine, and the customer will have to contact the bank to get it back.

If a transaction fails for any reason other than an invalid PIN, the ATM will display an explanation of the problem, and will then ask the customer whether he/she wants to do another transaction.

The ATM provides the customer with a printed receipt for each successful transaction, showing the date, time, machine location, type of transaction, account(s), amount, and ending and available balance(s) of the affected account ("to" account for transfers).

The ATM has a key-operated switch that allows an operator to start and stop the service to customers. After turning the switch to the "on" position, the operator is required to verify and enter the total cash on hand. The machine can only be turned off when it is not serving a customer. When the switch is moved to the "off" position, the machine will be shut down so that the operator may remove deposit envelopes and reload the machine with cash, blank receipts, etc.

The ATM also maintains an internal log of transactions to facilitate resolving ambiguities arising from a hardware failure in the middle of a transaction. Entries will be made in the log when the ATM is started up and shut down for each message sent to the Bank (along with the response back if one is expected), for the dispensing of cash, and for the receiving of an envelope. Log entries may contain card numbers and dollar amounts. But for the security reason it will never contain a PIN.

5.2.2 Creating an Initial Use Case Model

According to the requirements described above, we use the tool to define an initial use case model of the ATM system in the following steps.

1. Create a new use case model. Add three use cases: "Withdraw", "Deposit" and "Inquire" and one actor "Customer". The actor "Customer" uses these three use cases. Figure 44 shows the initial model at the environment level.

2. Define goals for the actor "Customer". There are three goals: "Withdraw cash", "Deposit cash or checks" and "Inquire the account balance". Define the context for use cases "Withdraw", "Deposit" and "Inquire" with text description.

   For example, the context of the use case "Deposit" is described as follows: a deposit transaction requires the customer to choose the type of account (e.g. checking account) to deposit to from the menu of possible accounts, and to type in an amount on the keyboard. The transaction is initially sent to the bank to verify whether the ATM can accept a deposit from this customer to

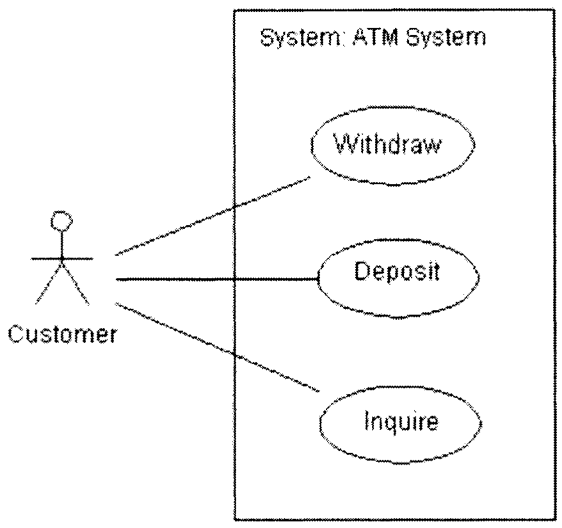


Figure 44: ATM Initial Use Case Model

this account or not. If the transaction is approved, the machine accepts an envelope from the customer containing cash and/or checks before it issues a receipt. Once the envelope is received, a second message is sent to the bank to confirm that the bank can credit the customer's account — contingent on the manual verification of the deposit envelope contents by an operator later.

3. Define episodes for use cases "Withdraw", "Deposit" and "Inquire" respectively. Their episode models are illustrated using episode trees as follows:

Episode model for the use case "Withdraw"

```
(sequence)
   1 Insert and validate the bank card
   2 Enter and validate PIN
   3 Select the bank account
   4 Enter the amount
   5 (alternation) ATM processes the withdrawal request
     5.1 (sequence) bank permits the operation
             5.1.1 Customer gets cash
             5.1.2 ATM updates the account balance
     5.2 Bank denies the operation
   6 ATM ejects the bank card
```

Episode model for the use case "Deposit"

```
(sequence)
   1 Insert and validate the bank card
   2 Enter and validate PIN
   3 Select the bank account
   4 Enter the amount
```

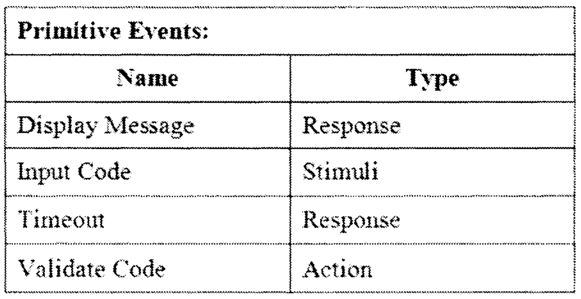
5 (alternation) ATM processes the deposit request
5.1 (sequence) bank permits the operation
5.1.1 Customer inserts the envelope
5.1.2 ATM updates the account balance
5.1.3 ATM prints the account information on the receipt
5.2 Bank denies the operation
6 ATM ejects the bank card

Episode model for the use case "Inquire"

(sequence)
1 Insert and validate the bank card
2 Enter and validate PIN
3 Select the bank account
4 (alternation) ATM processes the inquiry request
4.1 (sequence) bank permits the operation
4.1.1 ATM shows the account information on the screen
4.1.2 ATM prints the account information on the receipt
4.2 Bank denies the operation
6 ATM ejects the bank card

4. Define events for episodes above. Figure 45 shows a list of primitive events in episode "Enter and Validate PIN". As we can see, there are four events: "Display Message", "Input Code", "Timeout", "Validate Code". These events have different types: "Response", "Stimuli" and "Action".

   Since use case refactorings in the thesis are mainly at the environment level and structure level, in this case study we skip details at the event level.



Primitive Events:	
Name	Type
Display Message	Response
Input Code	Stimuli
Timeout	Response
Validate Code	Action

Figure 45: Primitive Events in the Episode "Enter and Validate PIN"

5.2.3 Evolving the Initial Model with Refactorings and Extensions

This section illustrates how the above simple initial model is evolved based on additional requirements. The evolution process involves use case refactorings which improve the structure of the use case model, and extensions which introduce new requirements. We describe the process in several major steps. We also introduce the motivation for each major step and related changes to the use case model.

Step 1. Add the "Do Transaction" use case and move shared episodes.

Motivation:

Use cases "Withdraw", "Deposit" and "Inquire" are bank transactions. They share some common episodes. It is preferable to create a parent use case "Do Transaction" to represent this nature. We also create the "Operation" service to package these use cases.

Detailed steps:

(1) Use the *Create_Empty_Service* refactoring to create the service "Operation".

(2) Move use cases "Withdraw", "Deposit" and "Inquire" into the service "Operation".

(3) Use the *Generalize_Usecases* refactoring to create a new use case "Do Transaction" in the service "Operation", set "Do Transaction" as the parent of use cases "Withdraw", "Deposit" and "Inquire", and move common episodes in use cases "Withdraw", "Deposit" and "Inquire" to use case "Do Transaction". These common episodes are: "Insert and validate the bank card", "Enter and validate PIN", and "Select the bank account".

Figure 46 shows the new use case model after adding the service "Operation" and the use case "Do Transaction".

Step 2. Add the approval process.

Motivation:

The following requirement is added to the use case model: due to security reasons, it needs a separate session for the customer's operations. The customer has to input his PIN to validate his identity within his session. The ATM system checks the PIN code to accomplish the approval process. If the PIN code is not correct, the exception will be handled within a transaction.

Detailed steps:

(1) Use the *Create_Empty_Usecase* refactoring to create use cases "Open Session", "Do Approval Process" and "Process Invalid PIN".

(2) Create the association between the actor "Customer" and the use case "Open Session".

(3) Use the *Create_Empty_Actor* refactoring to create the actor "Bank".

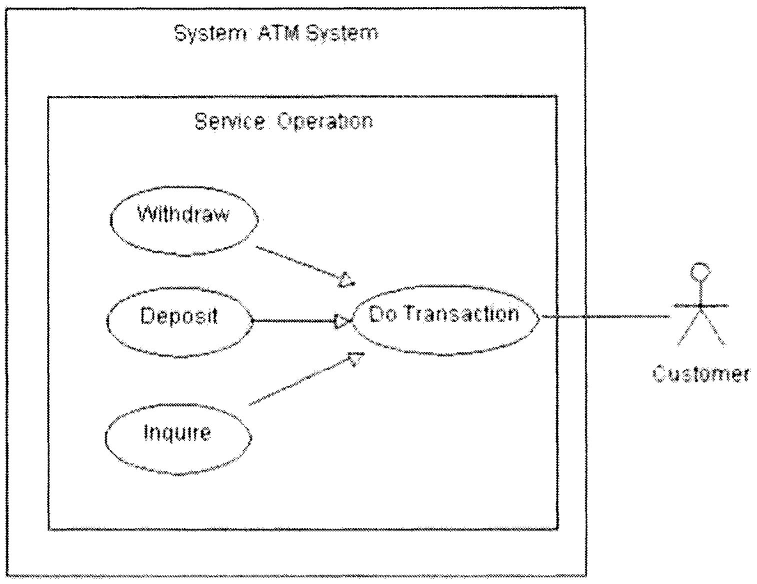


Figure 46: ATM Case Study: After Step 1

(4) Create the association between the actor "Bank" and the use case "Do Transaction".

(5) Remove the association between the actor "Customer" and the use case "Do Transaction".

(6) Remove associations between the actor "Customer" and use cases "Withdraw", "Deposit" and "Inquire".

Most of these steps are extensions that introduce new requirements. Figure 47 shows the new use case model after adding the approval process.

Step 3. Define the structure level.

Motivation:

Define the structure information including episodes, preconditions, context and postconditions of new use cases "Open Session", "Do Approval Process" and "Process Invalid PIN".

Step 4. Increase the reusability and remove the redundancy by making the use case "Open Session" to include use cases "Do Approval Process" and "Do Transaction", and making the use case "Process Invalid PIN" to extend the use case "Do Transaction".

Motivation:

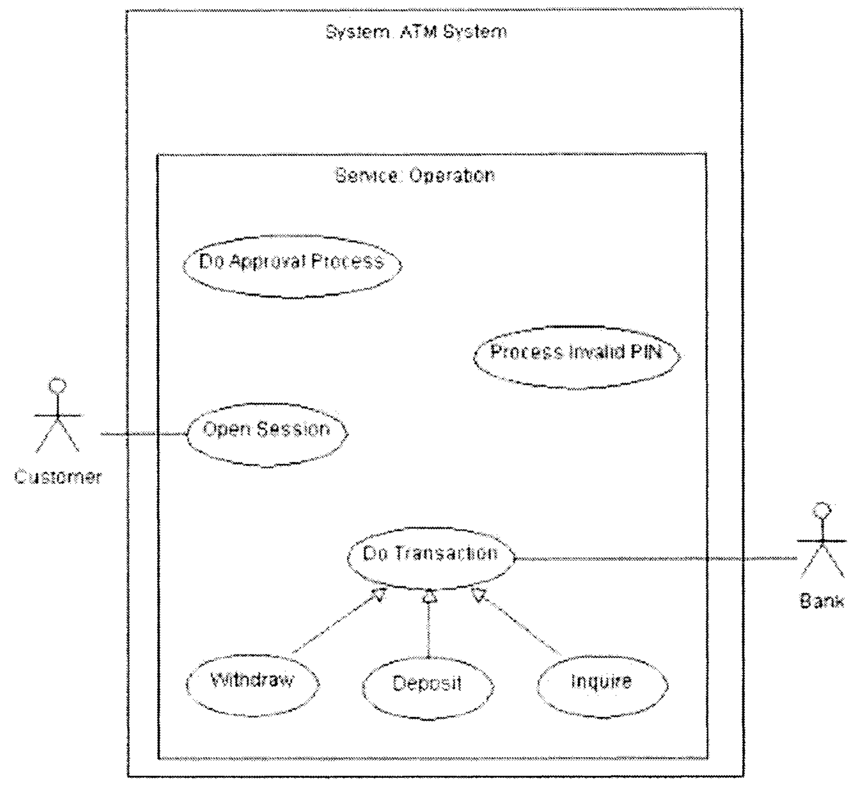


Figure 47: ATM Case Study: After Step 2

After the above structure level is defined, it is apparent that there are some relationships among use cases within the use case model. It is preferable for the use case model to reflect these relationships because it increases the understandability and reusability of the use case model.

Detailed steps:

(1) After examining episode models of use cases "Open Session" and "Do Transaction", it turns out that the episode tree of the use case "Do Transaction" is an episode subtree of the use case "Open Session". Therefore, an inclusion relationship is generated by making the use case "Open Session" to include the use case "Do Transaction" using two refactorings. A temporary use case is created by applying the *Usecase_Inclusion_Generation* refactoring to the use case "Open Session" so that this temporary use case is equivalent to the use case "Do Transaction". Then the *Reuse_Usecase* refactoring is applied to delete this temporary use case.

(2) Similar to the above step, after examining episode models of use cases "Open Session" and "Do Approval process", it turns out that the episode tree of the use case "Do

Approval process" is an episode subtree of the use case "Open Session". Therefore, an inclusion relationship is generated by making the use case "Open Session" to include the use case "Do Approval process" using the *Usecase_Inclusion_ Generation* and *Reuse_Usecase* refactorings.

(3) After examining episode models of use cases "Do Transaction" and "Process Invalid PIN", it turns out that the episode tree of the use case "Process Invalid PIN" is an episode subtree of the use case "Do Transaction". This episode subtree is an exceptional course. Therefore, an extension relationship is generated by making the use case "Process Invalid PIN" to extend the use case "Do Transaction" using two refactorings. A temporary use case is created by applying the *Usecase_Extension_Generation* refactoring to the use case "Do Transaction" so that this temporary use case is equivalent to the use case "Process Invalid PIN". Then the *Reuse_Usecase* refactoring is applied to delete this temporary use case.

These steps are all completed by refactorings. Figure 48 shows the new use case model after the above refactoring steps.

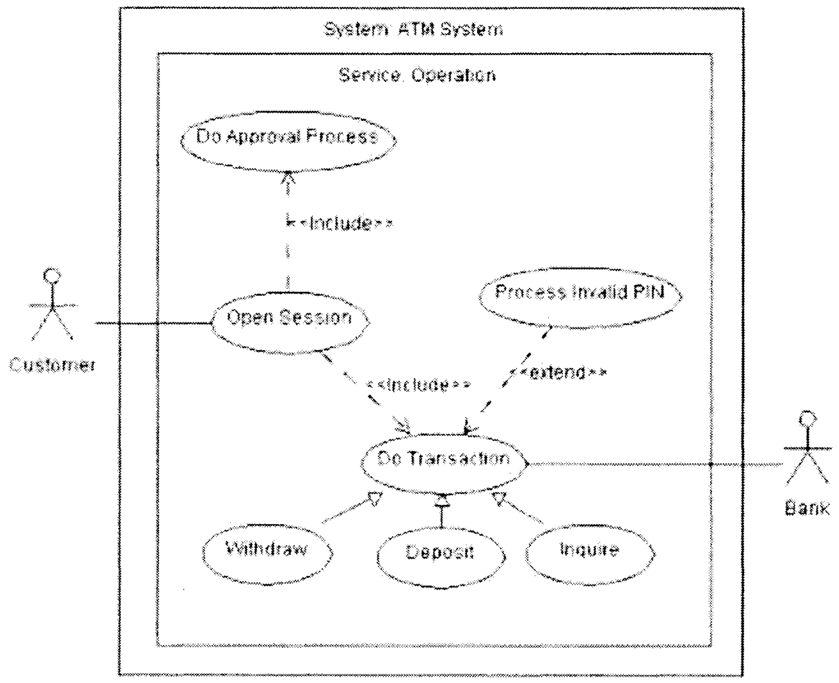


Figure 48: ATM Case Study: After Step 4

Step 5. Add the use case "Transfer".

Motivation:

Customers need to transfer funds between different accounts. We need to create a new use case "Transfer". It should support transactions. We also need to define its episodes, preconditions, context, postconditions and events.

Detailed steps:

(1) Use the *Create_Empty_Usecase* refactoring to create the use case "Transfer".

(2) Define preconditions, context and postconditions of the use case "Transfer".

(3) Define episodes of the use case "Transfer". Some episodes are already defined in the use case "Transaction".

(4) Define events of episodes in the use case "Transfer".

(5) Use the *Change_Parent_Usecase* refactoring to set the use case "Do Transaction" as the parent of the use case "Transfer". Figure 49 shows the new use case model after this refactoring.

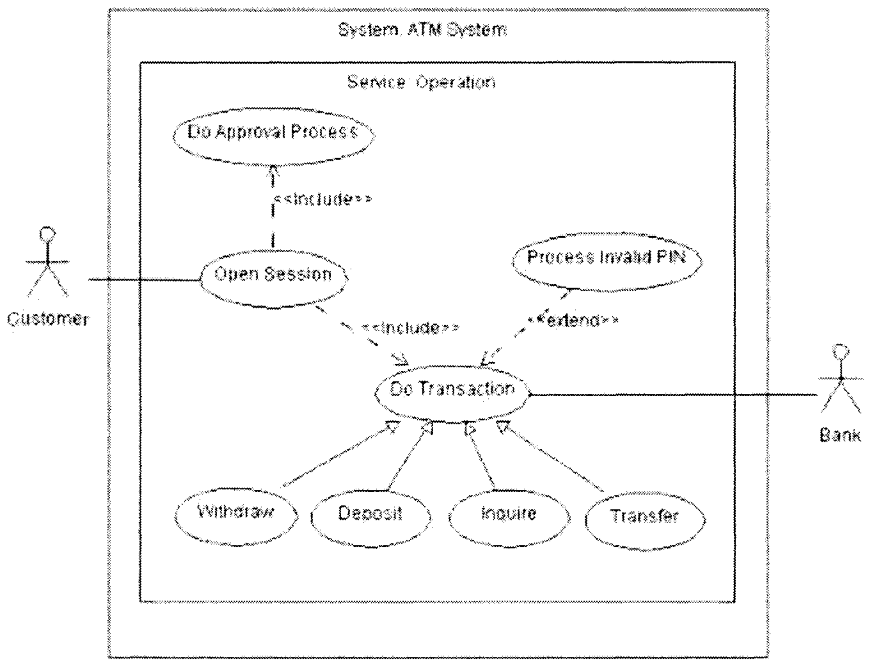


Figure 49: ATM Case Study: After Step 5

Step 6. Add the service *Maintenance*.

Motivation:

The ATM system needs to be maintained by the operator from the bank. We create use cases "Start System" and "Stop System" to represent this requirement. These use cases are grouped into a new service "Maintenance". We also define information at the structure and event level including episodes, preconditions, context, postconditions and events. We add the actor "Operator" to interact with these use cases.

Detailed steps:

(1) Use the *Create_Empty_Usecase* refactoring to create use cases "Start System" and "Stop System".

(2) Define episodes, preconditions, context and postconditions of use cases "Start System" and "Stop System".

(3) Define events of episodes in use cases "Start System" and "Stop System".

(4) Use the *Create_Empty_Actor* refactoring to create the actor "Operator".

(5) Define goals of the actor "Operator".

(6) Create associations between the actor "Operator" and use cases "Start System", "Stop System".

(7) Use the *Create_Empty_Service* refactoring to create the service "Maintenance".

(8) Move use cases "Start System" and "Stop System" into the service "Maintenance".

Figure 50 shows the new use case model after the above changes. This is also the final use case model of this case study.

5.3 Evaluation Conclusions

As mentioned at the beginning of this chapter, the purpose of these case studies is to evaluate the use case modeling and refactoring tool, validate use case refactorings, and investigate the whole refactoring process. This section analyses experimental results in detail.

5.3.1 Tool Evaluation

Modeling Functionalities

The tool provides functionalities to build use case models based on the use case metamodel defined in Section 3.1. In both case studies, the use case model is defined in three levels using the tool. At the environment level, use cases, actors, goals, services and related relationships are defined. At the

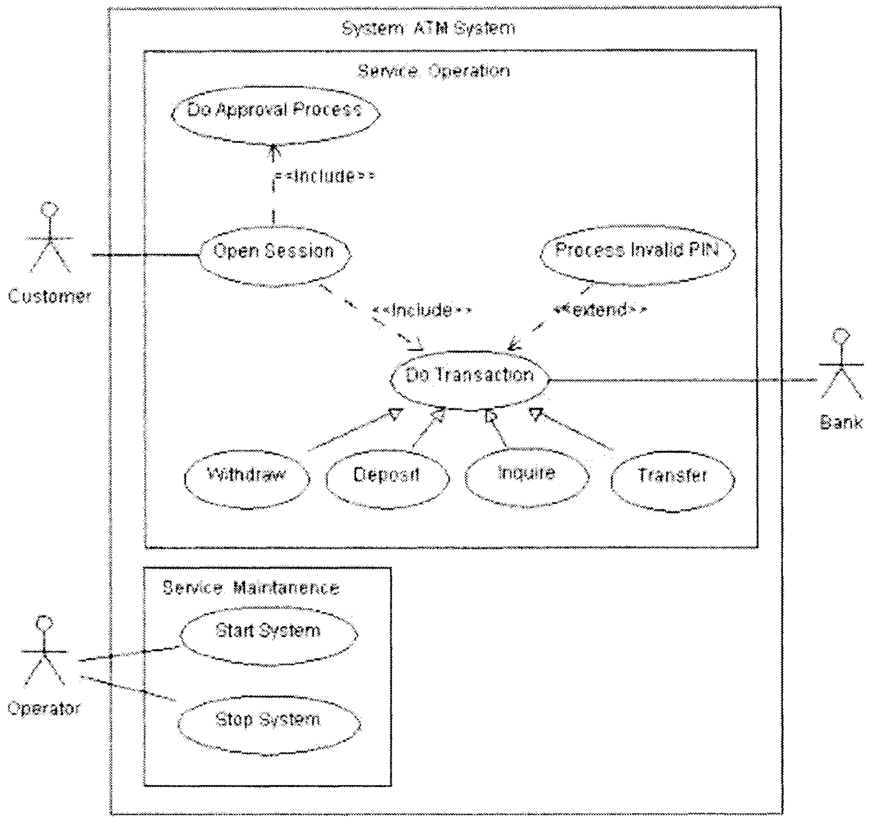


Figure 50: ATM Final Use Case Model

structure level, the context and episode model are defined. The episode model is shown as an episode tree. Each episode is unique in the use case model. Once it is defined, it can be used by all use cases within the model. At the event level, each event is unique in the use case model. Once it is defined, it can be used by all episodes within the model. The event model is defined for some episodes, but not for all of them. This is because use case refactorings in this research are mainly defined for the environment level and structure level. The tool allows to navigate the use case across different levels conveniently.

Refactoring Functionalities

In the VSS case study, there are seven refactoring steps involved, including five distinct use case refactorings. In the ATM case study, there are seventeen refactoring steps involved, including eight distinct use case refactorings. The tool checks preconditions of each refactoring step automatically. It applies the refactoring to the use case model and makes sure that postconditions are met for each refactoring. The tool performs these changes automatically. It satisfies basic needs in the refactoring process. Admittedly, it will be nice if some of these refactoring steps can be executed in a batch model using scripts or macros.

5.3.2 Validation of Use Case Refactorings

Ten use case refactorings are used in total within these two case studies. These ten refactorings are:

- *Create_Empty_Usecase*
- *Create_Empty_Actor*
- *Create_Empty_Service*
- *Generalize_Usecases*
- *Usecase_Inclusion_Generation*
- *Usecase_Extension_Generation*
- *Usecase_Equivalence_Generation*
- *Reuse_Usecase*
- *Change_Parent_Usecase*
- *Change_Parent_Actor*

After each refactoring step, the behavior of the use case model is doubly checked manually to ensure the behavior preservation. In both case studies, all of these ten refactorings preserve the behavior of the model.

5.3.3 The Refactoring Process

VSS Case Study

In the VSS case study, there are seven refactoring steps involved, which are:

- Use the *Change_Parent_Actor* refactoring to establish a generalization relationship between actors "Staff" and "Manager Staff".
- Use the *Change_Parent_Actor* refactoring to establish a generalization relationship between actors "Customer" and "Dependent Customer".
- Use the *Usecase_Inclusion_Generation* refactoring to break up the use case "Web Video Inquiry" into new use cases "Web Video Inquiry" and "Video Search".

- Use the *Usecase_Equivalence_Generation* refactoring to make the use case "Video Search" equivalent to the use case "Video Inquiry".
- Use the *Reuse_Usecase* refactoring to remove the use case "Video Search".
- Use the *Generalize_Usecases* refactoring to create the use case "Video Maintenance".
- Use the *Generalize_Usecases* refactoring to create the use case "Customer Maintenance".

After these refactoring steps, a new use case model is generated as shown in Figure 43. Compared to the initial model as shown in Figure 37, the structure of the use case model is greatly improved. The new use case model reduces the redundancy by making the use case "Web Video Inquiry" to include "Video Inquiry". It groups use cases "Add Video", "Delete Video", "Update Video" into the video maintenance category and use cases "Add Customer", "Delete Customer", "Update Customer" into the customer maintenance category. Thus the new use case model reveals the inherent nature of these use cases. These refactoring steps improve the reusability, understandability and maintainability of the use case model.

ATM Case Study

In the ATM case study, there are seventeen refactoring steps involved, which are:

- Use the *Create_Empty_Service* refactoring to create the service "Operation".
- Use the *Generalize_Usecases* refactoring to create a new use case "Do Transaction" in the service "Operation", set "Do Transaction" as the parent of use cases "Withdraw", "Deposit" and "Inquire", and move common episodes in use cases "Withdraw", "Deposit" and "Inquire" into the use case "Do Transaction".
- Use the *Create_Empty_Usecase* refactoring to create use cases "Open Session", "Do Approval Process" and "Process Invalid PIN".
- Use the *Create_Empty_Actor* refactoring to create the actor "Bank".
- Apply the *Usecase_Inclusion_Generation* refactoring to the use case "Open Session" so that the newly created temporary use case is equivalent to the use case "Do Transaction".
- Use the *Reuse_Usecase* refactoring to delete the above temporary use case.
- Apply the *Usecase_Inclusion_Generation* refactoring to the use case "Open Session" so that the newly created temporary use case is equivalent to the use case "Do Approval process".
- Use the *Reuse_Usecase* refactoring to delete the above temporary use case.

- Apply the *Usecase_Extension_Generation* refactoring to the use case "Do Transaction" so that the newly created temporary use case is equivalent to the use case "Process Invalid PIN".
- Use the *Reuse_Usecase* refactoring to delete the above temporary use case.
- Use the *Create_Empty_Usecase* refactoring to create the use case "Transfer".
- Use the *Change_Parent_Usecase* refactoring to set the use case "Do Transaction" as the parent of the use case "Transfer".
- Use the *Create_Empty_Usecase* refactoring to create the use case "Start System".
- Use the *Create_Empty_Usecase* refactoring to create the use case "Stop System".
- Use the *Create_Empty_Actor* refactoring to create the actor "Operator".
- Use the *Create_Empty_Service* refactoring to create the service "Maintenance".

In this case study, the final use case model as shown in Figure 50 is created incrementally starting from a simple model as shown in Figure 44. Starting with a simple use case model, we improve the structure of the use case model by refactorings. Then new requirements are added into the use case model. After adding new requirements, the structure of the use case model is changed, which may bring needs to improve the structure by refactorings again. Thus the whole evolution process is viewed as refactorings plus extensions.

Refactoring Guidelines

Generally, a use case model can be created incrementally in the follow steps:

(1). Analyse the requirements and get a brief description.

(2). Define use cases, actors and users.

(3). Define goals of actors or tasks of users.

(4). Define the structure information, including the episode, precondition and postcondition.

(5). Define events of episodes.

(6). Apply use case refactorings to improve the structure of the use case model so that additional requirements can be added more easily.

(7). Repeat Step (2) – Step (6) until the use case model meets all the requirements.

In practice, sometimes it may be hard to know when to apply use case refactorings. Here are some general guidelines.

- Redundant Definitions: If the episode subtree is redundantly defined, it may be preferable to apply refactoring to remove the redundancy.

  Examples:
  (1). VSS case study: The redundancy is removed by making the use case "Web Video Inquiry" to include "Video Inquiry".

  (2). ATM case study: step 4 - make the use case "Open Session" to include use cases "Do Approval Process" and "Do Transaction", and make the use case "Process Invalid PIN" to extend the use case "Do Transaction".

- Hidden Generalization Relationship: If there is an inherent generalization relationship between use cases or actors, it is preferable to apply refactorings to reveal the generalization relationship explicitly.

  Examples:
  (1). VSS case study: It groups use cases "Add Video", "Delete Video", "Update Video" into the video maintenance category and use cases "Add Customer", "Delete Customer", "Update Customer" into the customer maintenance category.

  (2). ATM case study: Step 1 - add the "Do Transaction" use case and move shared episodes.

  (3). ATM case study: Step 5 - use the *Change_Parent_Usecase* refactoring to set the use case "Do Transaction" as the parent of the use case "Transfer".

- Refactoring for New Requirements: It is quite often that refactorings are driven by new requirements. If refactorings will make it easier to address new requirements, it is preferable to apply them.

  Examples:
  (1). ATM case study: After adding the use case "Do Transaction" by refactorings, it is more convenient to add the use case "Transfer" later.

Similar to the source code refactoring, there are times when you should not refactor at all. When it is close to a deadline, the gain from refactoring would appear after the deadline and thus be too late. The other time is when the model is such a mess that it is easier to start from scratch instead of refactoring the existing model.

Chapter 6

Conclusion and Future Work

This chapter concludes the thesis by answering some fundamental research questions. It compares this research with some related work and discusses major contributions of this research. It also illustrates its limitations and discusses future work.

6.1 Conclusion

Some fundamental research questions have formed the basis of this work. This section answers these questions by summarizing the work presented in the thesis.

Research Question 1: Is it necessary to restructure use case models?

The importance of the source code refactoring has been increasingly well understood. As more and more mainstream commercial tools provide capabilities for refactorings, source code refactoring has become part of many people's routine work. Because refactoring use case models is a totally new approach, it is not surprising that its necessity is questioned. However, the same motivation is behind refactoring use case models as refactoring source code. During software evolution, requirements change. It is a challenge to update use case models consistently, especially for the large-scale complex system. Restructuring use case models is an effective way to improve the understandability and maintainability. In fact, as introduced in Section 2.3, there are many other research efforts on how to transform models. As maintaining models becomes an increasingly time-consuming activity, people will be more aware of the importance of these transformations.

Research Question 2: What does the behavior preservation mean when refactoring use case models?

Compared to the source code refactoring, it is much more complicated to define and to verify the behavior preservation in the use case refactoring. The program is executable. It is possible to check if the output values remain the same when the program is called twice (once before and once after a refactoring). For the use case refactoring, however, it is difficult and controversial to argue the issue of behavior preservation. The presented work addresses this issue in several steps. Firstly, it defines a use case metamodel. This metamodel forms a basis for the discussion of the behavior preservation because it defines the meaning of use case entities and related relationships. The behavior of a use case is defined by its episode model, that is, the behavior is not only defined by a set of episodes, but how these episodes are related to each other to form the internal structure of the use case. Similarly, the behavior of an episode is defined by its event model. Secondly, it introduces a process algebra semantics for the use case model. The behavior of a use case can be defined by a process algebra expression based on the episode model and event model. The behavior of a use case model is defined in terms of the individual use cases and their relationships. Similar to the source code refactoring, the presented work also defines invariants so that the refactored use case model is syntactically correct and semantically equivalent to the model before refactoring.

Research Question 3: How are use case refactorings validated?

In the refactoring definition, each refactoring is verified against the definition of behavior preservation. In order to validate use case refactorings, a use case modeling and refactoring tool is developed, which makes it possible to create the use case model based on the metamodel. The tool also provides the automatic support for applying refactorings. Two case studies are used to further validate use case refactorings. For each refactoring step in these case studies, the behavior of the use case model before and after refactoring is compared to ensure that the refactoring does not change the behavior. The behavior preserving property of each refactoring will become more convincing when more and more case studies are conducted.

Research Question 4: Is refactoring use case models feasible and practical?

The thesis defines fifty-three use case refactorings. These refactorings can be used to compose/decompose a use case, compose/decompose an episode, and to generalize use cases, actors and episodes. These transformation operations are heavily used during the evolution of use case models. For example, the size of each use case must not be too large. Otherwise, it is difficult to understand. However, a too large collection of small use cases is also not desirable due to the lack of overview. Refactorings on composing/decomposing a use case can avoid these undesired models.

Two case studies are carried out to evaluate some of these refactorings under the tool support. In the ATM case study, the use case model is built incrementally with use case refactorings starting from a simple model. This case study shows how use case refactorings can improve the structure of the use case model so that new requirements can be added more easily. In the VSS case study, it starts with a relatively more complex use case model. This case study shows how the structure can be improved so that the use case model becomes more understandable and maintainable. The presented work shows clearly that refactoring use case models is feasible and practical. It also provides some guidelines on the refactoring process.

6.2 Contributions

This work makes the following major contributions. These contributions represent key milestones within this research.

1. A three-level use case metamodel with a hierarchical structure.

   This metamodel is defined to formalize use case terminologies. It is a foundation for describing the behavior of the use case model. This metamodel is defined to address different needs in requirements engineering and software engineering.

   This metamodel is based on Regnell's use case model [REG99]. The three-level hierarchical structure is borrowed from Regnell's model. However, several major improvements are made in this metamodel. Firstly, it defines use case relationships and actor relationships. This metamodel covers use case relationships defined in Objectory [JAC92], SOMA [SEL97], OML [FIR98] and UML (v2.0) [UML20]. It defines the "generalization" relationship between actors so that their commonalities can be modeled.

   Secondly, it refines Regnell's model at the structure level by introducing the episode model and episode tree, which describe not only primitive episodes but also their relationships. This helps manage and present the use case internal structure effectively. Based on the episode model, the use case model can be restructured by manipulating episodes among different use cases. Similarly, this metamodel also defines the event model and event tree.

   Compared with the use case metamodel defined in the UML [UML20], this metamodel specifies the use case behavior using a three-level hierarchical structure. While the environment level presents a similar overview with the one in the UML, the structure level and event level define the use case behavior in detail. Use cases in the UML are specified with the text description, which makes the use case behavior vague. They are used to illustrate the functionality of the system from the user's point of view. The UML uses the dynamic model to reveal the internal behavior of the system. The dynamic model includes *sequence diagrams*, *activity diagrams*, and *state machine diagrams*. The UML does not have a formally defined semantics.

2. A process algebra semantics for the use case model.

Based on the use case metamodel, the thesis defines a process algebra semantics for the use case model. Use case semantics is defined as the execution of the use case. The semantics of the use case model is defined as the set of possible executions of all use cases within the model. The semantics of a single episode is specified using process algebra PA_{BMSC} [MAUW94]. The semantics of a single use case is defined in terms of the episode model, which describes a list of primitive episodes as well as their relationships. The thesis analyses different use case relationships and evaluates the behavior of the use case model. It handles the semantics issue by modeling the whole system as a single use case. Use cases with different relationships are mapped into this single use case so that the semantics of the use case model can be defined based on the episode model. Thus the semantics of the use case model can be defined in terms of the individual use cases and their relationships.

This semantics makes it possible to verify the behavior preservation of the use case refactoring. It is the foundation for refactoring use case models. As some people in the industry transforms use case models in their own ways, this semantics helps them understand related impact on the behavior of the use case model. For example, John McGregor et al. develope *use case assortment* for the requirement analysis phase of framework development [MIL99]. This approach identifies commonality and variability among use cases within the use case model. To reflect the commonality/variability analysis, the use case model is factored by introducing abstract use cases and abstract actors to rearrange responsibilities. These transformation steps can be justified with the presented process algebra semantics for the use case model.

There is some related work on defining use case semantics. Perdita Stevens uses a Labeled Transition System (LTS) to formally interpret use cases as sequences of actions in [STV01]. Butler et al. [BUT97] give a specification of use cases in the Z notaion [SPI92]. The focus is on understanding use cases, not on instrumenting them for the specification in combination with the Z notation. Grieskamp et al. [GRI00] formalize use cases using the Z notation. They extend this work and specify use cases in Abstract State Machine Language [GRI01]. Dranidis et al. [DRA03] propose the specification of use cases with X-machines [EIL74]. They present a method for transforming use case text into its corresponding X-machine model.

3. A list of fifty-three use case refactorings.

The thesis is the first literature to define a comprehensive list of use case refactorings. Although the concept of refactoring use case models is proposed in Cascaded Refactoring [BUT01] and about eight use case refactorings is defined further in [XU06], it is hard to verify the behavior preservation of these refactorings due to the lack of the use case semantics. The thesis defines fifty-three refactorings using a template. Each refactoring is described in terms of the refactoring description, arguments, preconditions, postconditions, and the verification of the behavior preservation. These refactorings are divided into three levels: environment level, structure level, and event level.

These refactorings provide various transformation capabilities including generalization of use cases and actors, decomposition of use cases and episodes, and composition of use case relationships and episode relationships. These transformation operations are routine tasks performed by people to reorganize the use case model. These refactorings are very useful and effective in improving the structure of the use case model.

Compared with the source code refactoring [OPD92], the thesis uses a similar format to define use case refactorings. Unlike the program, the use case model is not executable, It is more difficult to verify the behavior preservation property of each use case refactoring. The thesis identifies syntactic and semantic properties required to preserve the behavior of the use case model.

4. A use case modeling and refactoring tool.

   This tool provides an integrated environment for creating the use case model based on the above use case metamodel and for refactoring the use case model. It facilitates the refactoring process. It allows to check refactoring preconditions and to apply the refactoring automatically.

   Source code refactorings are usually performed with tool support. The presented use case modeling and refactoring tool enables that use case refactorings can be applied automatically. This avoids the error-prone manual process and improves the efficiency of the refactoring process.

5. Validation of use case refactorings with the tool support using two case studies.

   These case studies demonstrate that it is feasible and practical to evolve the use case model starting from a simple model and to improve the structure of the more complex use case model by refactorings. The thesis investigates the evolution process of use case models. It provides some guidelines on the refactoring process.

   Compared with the source code refactoring, it is much more difficult to carry out case studies of refactoring use case models in the industry. Use case refactorings are based on the presented use case metamodel. However, this metamodel is not yet accepted by the industry. The thesis presents two case studies to simulate two kinds of industrial projects. The ATM case study is used to demonstrate how to build the use case model incrementally with refactorings and extensions. This simulates the stage of constructing use case models. The VSS case study is used to demonstrate how the structure of the more complex use case model can be improved by refactorings step by step. This simulates the maintenance of use case models.

6.3 Future Work

There is much debate within the community on how to formalize use cases. The thesis attempts to formalize use cases by defining a use case metamodel. A process algebra semantics is defined based on this metamodel. This makes it possible to verify the behavior preserving property of each refactoring. However, this formalization is still in its infancy. The metamodel may need to be changed to satisfy different needs. For the purpose of simplicity, this semantics does not provide support for all concepts. The precondition and postcondition of a use case, for example, are not covered in this semantics.

In the metamodel, we introduce concepts of goal and task. However, the detailed application of goal and task is not investigated in this research. This will be addressed in future research.

During the software evolution, the requirements may not be clear. The use case model may not be fully specified. This makes it difficult to verify the behavior preservation of the use case refactoring. The thesis defines a set of invariants that needs to be preserved during refactoring. Each refactoring is verified against the behavior preservation in an informal way. Ideally, the correctness of each refactoring should be formally proved.

In the thesis, some refactorings are defined in a restrictive way to ensure that the use case model remains semantically equivalent across refactorings. For example, the *Split_Usecase* refactoring is defined under the condition that the use case is not referenced by any other use case. However, the behavior can be preserved under less restrictive conditions, which will permit some useful refactoring operations. This issue is to be addressed in future research.

Refactoring use case models is a totally new approach in the community. Although the thesis presents two case studies, there is still a lot to do. More case studies are needed to further validate use case refactorings. While knowledge from the process of the source code refactoring is useful, the use case refactoring process needs to be studied further. This will improve the understanding to the whole process. Similar to source code refactorings, it is a continuous effort to discover new use case refactorings.

The use case refactoring tool also needs further improvements. More refactorings should be implemented. Some features in source code refactoring tools are very useful. For example, the tool should provide the "undo" functionality to undo the use case refactoring. It should allow refactorings to be executed in a batch mode using scripts or macros.

Bibliography

[ACE01] L. Aceto, W. J. Fokkink and C. Verhoef. *Structural Operational Semantics*. Handbook of Process Algebra, North-Holland, Amsterdam, p197–292, 2001.

[ADO03] W. S. Adolph and P. Bramble. *Patterns for Effective Use Cases*. Addison Wesley, 2003. ISBN: 0201721848.

[ALUR96] R. Alur, G. J. Holzmann and D. Peled. *An Analyzer for Message Sequence Charts*. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96), volume 1055 of Lecture Notes in Computer Science, p35–48, Passau, Germany, 1996. Springer Verlag.

[AND97] M. Andersson and J. Bergstrand. *Formalizing Use Cases with Message Sequence Charts*. Master thesis, Department of Communication Systems at Lund Institute of Technology, 1997.

[ANN67] J. Annett and K. Duncan. *Task Analysis and Training Design*. Occupational Psychology 41, p211–221, 1967.

[ANT97] A. I. Anton. *Goal Identification and Refinement in the Specification of Software-Based Information Systems*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, June 1997.

[BAE04] J. C. M. Baeten. *A Brief History of Process Algebra*. Rapport CSR 04–02, Vakgroep Informatica, Technische Universiteit Eindhoven, 2004.

[BAE90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, Cambridge, 1990.

[BAN87] J. Banerjee and W. Kim. *Semantics and Implementation of Schema Evolution in Object-Oriented Databases*. In Proceedings of the ACM SIGMOD Conference, 1987.

[BECK99] K. Beck. *Extreme Programming Explained: Embracing Change*. Addison Wesley, 1999.

[BEEK94] M. von der Beek. *A Comparison of Statechart Variants*. In W.-P. de Roever H. Langmaack and J. Vytopil, editors, Formal Techniques in Real-Time and Fault-Tolerant Systems, number 863 in Lecture Notes in Computer Science, p128–148. Springer Verlag, September 1994.

[BEK71] H. Bekič. *Towards a Mathematical Theory of Processes.* Technical Report TR 25.125, IBM Laboratory Vienna, 1971.

[BEK84] H. Bekič. *Programming Languages and Their Definition (Selected Papers edited by C.B. Jones).* Number 177 in LNCS. Springer Verlag, 1984.

[BEN00] K. H. Bennett and V. T. Rajlich. *Software Maintenance and Evolution.* ICSE '00: Proceedings of the Conference on The Future of Software Engineering, p73–87. ACM Press, 2000. ISBN: 1-58113-253-0.

[BER82] J. A. Bergstra and J. W. Klop. *Fixed Point Semantics in Process Algebra.* Technical Report IW 208, Mathematical Centre, Amsterdam, 1982.

[BER84] J. A. Bergstra and J. W. Klop. *Process Algebra for Synchronous Communication.* Information and Control, 60(1/3): p109–137, 1984.

[BERG99] K. G. van den Berg and A. J. H. Simons. *Control-Flow Semantics of Use Cases in UML.* Information and Software Technology, 41(10), p651–659, 1999.

[BEST01] E. Best, R. Devillers and M. Koutny. *A Unified Model for Nets and Process Algebras.* Handbook of Process Algebra, North-Holland, Amsterdam, p945–1045, 2001.

[BJO] R. C. Bjork. *ATM Simulation*, Gordon College. http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/.

[BOG02] M. Boger, T. Sturm and P. Fragemann. *Refactoring Browser for UML.* Proceedings 3rd Int'l Conf. on eXtreme Programming and Flexible Processes in Software Engineering, p77–81, 2002.

[BOH96] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press, 1996.

[BOR00] E. Börger, A. Cavarra and E. Riccobene. *An ASM Semantics for UML Activity Diagrams.* In Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST), Iowa City, IO, USA, May 2000. Springer Verlag.

[BORL] Borland. http://www.codegear.com/tabid/102/Default.aspx.

[BOT03] P. Bottoni, F. Parisi-Presicce and G. Taentzer. *Specifying Integrated Refactoring with Distributed Graph Transformations.* In Proceedings of the Agtive Conference, 2003.

[BRO87] F. Brooks. *No Silver Bullet: Essence and Accidents of Software Engineering.* IEEE Computer, Vol. 20, No. 4, April, 1987.

[BUHR98] R. Buhr. *Use Case Maps as Architectural Entities for Complex Systems.* IEEE Transactions on Software Engineering, 24(12), p1131–1155, 1998.

[BUT01] G. Butler and L. Xu. *Cascaded Refactoring for Framework Evolution.* Proceedings of 2001 Symposium on Software Reusability. ACM Press, p51–57.

[BUT97] G. Butler, P. Grogono and F. Khendek. *A Z Specification of Use Cases.* In Proceedings of the Asia-Pacific Software Engineering Conference and International Computer Science Conference, p505–506. IEEE Computer Society Press, 1997.

[CAR95] J. M. Carroll. *Scenario–Based Design: Envisioning Work and Technology in System Development.* Wiley, New York, 1995.

[COC01] A. Cockburn. *Writing Effective Use Cases.* Addison Wesley, 2001. ISBN: 0201702258.

[COC97] A. Cockburn. *Structuring Use Cases with Goals.* Journal of Object Oriented Programming, 10(5) Sep. 97 and 10(7) Nov. 97.

[DAR93] A. Dardenne, A. Van Lamsweerde and S. Fickas. *Goal-Directed Requirements Acquisition.* Science of Computer Programming, 20(1–2), p3–50, April 1993.

[DAV93] A. Davis. *Software Requirements: Objects, Functions and States.* Prentice Hall PTR; 2 edition, 1993.

[DEVE] Developer Express. http://www.devexpress.com/Products/NET/IDETools/Refactor/.

[DON99] D. B. Roberts. *Practical Analysis for Refactoring.* PhD thesis, University of Illinois, 1999.

[DRA03] D. Dranidis, K. Tigka and P. Kefalas. *Formal Modelling of Use Cases with X-machines.* In Proceedings of South-East European Workshop on Formal Methods (SEEFM)'03, November 2003.

[ECLIPSE] Eclipse. http://www.eclipse.org/.

[EIL74] S. Eilenberg. *Automata, Languages and Machines.* Academic Press, New York, 1974.

[ESH00] R. Eshuis and R. Wieringa. *Requirements Level Semantics for UML Statecharts.* Formal Methods for Open Object-Based Distributed Systems IV, p121–140. Kluwer Academic Publishers, 2000.

[ESH01] R. Eshuis and R. Wieringa. *A Formal Semantics for UML Activity Diagrams — Formalising Workflow Models.* Technical Report CTIT–01–04, University of Twente, Department of Computer Science, 2001.

[FIR98] D. G. Firesmith, B. Henderson-Sellers and I. Graham. *OPEN Modeling Language (OML) Reference Manual.* Cambridge University Press, New York, 1998. ISBN: 0521648238.

[FLO67] R. W. Floyd. *Assigning Meanings to Programs.* Proceedings Symposium in Applied Mathematics, Mathematical Aspects of Computer Science, p19–32, AMS, 1967.

[FOW97] M. Fowler. *Analysis Patterns: Reusable Object Models.* Addison Wesley, 1997. ISBN: 0201895420.

[FOW99] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 1999.

[FRA03] R. France, S. Ghosh, E. Song and D. K. Kim. *A Metamodeling Approach to Pattern-Based Model Refactoring.* IEEE Software, Special Issue on Model Driven Development, Vol.20, No.5, p52–58, September/October 2003.

[GAM95] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software.* Addison Wesley, Reading, Massachusetts, 1995.

[GEH98] T. Gehrke, U. Goltz and H. Wehrheim. *The Dynamic Models of UML: Towards a Semantics and its Application in the Development Process.* Hildesheimer Informatik-Bericht 11/98, 1998.

[GHE98] C. Ghezzi and B. Nuseibeh. *Managing Inconsistency in Software Development.* Transactions on Software Engineering, 24(11): p906–907, 1998.

[GLA01] R. J. van Glabbeek. *The Linear Time — Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes.* Handbook of Process Algebra, North-Holland, Amsterdam, p3–100, 2001.

[GLA96] R. J. van Glabbeek and W. P. Weijland. *Branching Time and Abstraction in Bisimulation Semantics.* Journal of the ACM, 43:p555–600, 1996.

[GORP03] P. V. Gorp, H. Stenten, T. Mens and S. Demeyer. *Towards Automating Source — Consistent UML Refactorings.* Proceedings. UML 2003.

[GRA93] J. Grabowski, P. Graubmann and E. Rudolph. *Towards a Petri Net Based Semantics Definition for Message Sequence Charts.* In Proceedings of the Sixth SDL Forum, p179–190, Darmstadt, 1993. Amster-dam, North-Holland.

[GRI00] W. Grieskamp and M. Lepper. *Using Use Cases in Executable Z.* In IEEE Conference on Formal Engineering Methods, p111–120, 2000.

[GRI01] W. Grieskamp, M. Lepper and W. Schulte. *Testable Use Cases in the Abstract State Machine Language*. In Proceedings of the Second Asia-Pacific Conference on Quality Software, 2001.

[HA87] D. Harel. *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming, 8: p231–274, 1987.

[HAR87] D. Harel, A. Pnueli, J. P. Schmidt and R. Sherman. *On the Formal Semantics of Statecharts*. In Proceedings 2nd IEEE Symposium on Logic in Computer Science, p56–64. IEEE Press, 1987.

[HAR96] D. Harel and A. Naamad. *The STATEMATE Semantics of StateCharts*. ACM Transactions on Software Engineering, 5(4): p293–333, 1996.

[HAS05] J. Hassine, J. Rilling and R. Dssouli. *An Abstract Operational Semantics for Use Case Maps*. In Proceedings of Requirement Engineering 2005 (13th IEEE International Requirement Engineering Conference), Paris, France, September 2005. IEEE CS Press, p467–468.

[HAU05] Ø. Haugen. *Comparing UML 2.0 Interactions and MSC-2000*. Lecture Notes in Computer Science, Volume 3319, p65–79, Springer Berlin, 2005.

[HEY00] S. Heymer. *A Semantics for MSC Based on Petri Net Components*. SAM 2000, 2nd Workshop on SDL and MSC, Col de Porte, Grenoble, France, June 26–28, 2000.

[HOA69] C. A. R. Hoare. *An Axiomatic Basis for Computer Programming*. Communications of the ACM, 12: p576–580, 1969.

[HOA78] C. A. R. Hoare. *Communicating Sequential Processes*. Communications of the ACM, 21(8): p666–677, 1978.

[HOO92] J. J. M. Hooman, S. Ramesh and W-P. de Roever. *A Compositional Axiomatization of Statecharts*. Theoretical Computer Science, 101: p289–335, 1992.

[HOP01] J. E. Hopcroft, R. Motwani and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Second Edition, Addison Wesley, 2001.

[HUI91] C. Huizing. *Semantics of Reactive Systems: Comparision and Full Abstraction*. PhD thesis, Technical University Eindhoven, 1991.

[HUIZ91] C. Huizing and W. P. de Roever. *Introduction to Design Choices in the Semantics of Statecharts*. Information Processing Letters, 37: 205–213, February 1991.

[HUR97] R. R Hurlbut. *A Survey of Approaches for Describing and Formalizing Use Cases*. http://www.iit.edu/ rhurlbut/xpt-tr-97-03.html.

[IDEA] Ideat Solutions. http://www.refpp.com/.

[JAC05] I. Jacobson and P. W. Ng. *Aspect–Oriented Software Development with Use Cases.* Addison Wesley, 2005.

[JAC92] I. Jacobson, G. Booth, P. Jonsson and G. Overgaard. *Object-Oriented Software Engineering: a Use-Case Driven Approach.* Addison Wesley, 1992.

[JAC94] I. Jacobson, M. Ericsson and A. Jacobson. *The Object Advantage: Business Process Reengineering with Object Technology.* ACM Press, 1994, ISBN: 0-201-42289-1.

[JAC95] I. Jacobson. *Modeling with Use Cases: Formalizing Use-Case Modeling.* Journal of Object Oriented Programming, June 1995, Vol. 8 No. 3.

[JAC99] I. Jacobson, G. Booch and J. Rumbaugh. *The Unified Software Development Process.* Addison Wesley, Longman, 1999.

[JOH92] P. Johnson. *Human–Computer Interaction: Psychology, Task Analysis and Software Engineering.* McGraw-Hill, London, 1992. ISBN: 0077072359.

[JON01] B. Jonsson and G. Padilla. *An Execution Semantics for MSC2000.* 10th International SDL Forum Copenhagen, Denmark, June 2001. Springer Verlag LNCS 2078.

[JREF] JRefactory. http://jrefactory.sourceforge.net/.

[KRU03] P. Kruchten. *The Rational Unified Process: An Introduction.* Third Edition, Addison Wesley, 2003.

[LAD92] P. B. Ladkin and S. Leue. *An Analysis of Message Sequence Charts.* Technical Report IAM-92-013, University of Berne, 1992.

[LAD95] P. B. Ladkin and S. Leue. *Interpreting Message Flow Graphs.* Formal Aspects of Computing, 7(5): p473–509, 1995.

[LED02] H. Ledang and J. Souquières. *Integration of UML and B Specification Techniques: Systematic Transformation from OCL Expressions into B.* APSEC, p495, 2002.

[LI04] X. Li, Z. Liu and J. He. *A Formal Semantics of UML Sequence Diagram.* 15th Australian Software Engineering Conference (ASWEC 2004), April 2004, Melbourne, Australia. Australian Software Engineering Conference, IEEE Computer Society, 2004, ISBN: 0-7695-2089-8.

[LUT99] G. Lüttgen, M. von der Beeck and R. Cleaveland. *Statecharts via Process Algebra.* 10th International Conference on Concurrency Theory (CONCUR '99), volume 1664 of Lecture Notes in Computer Science, p399–414, Eindhoven, The Netherlands, August 1999. Springer Verlag.

[MAUW94] S. Mauw and M. Reniers. *An Algebraic Semantics of Basic Message Sequence Charts.* The Computer Journal, 37(4): p269–277, 1994.

[MCC63] J. McCarthy. *A Basis for a Mathematical Theory of Computation.* Computer Programming and Formal Systems, p33–70. North-Holland, Amsterdam, 1963.

[MENS04] T. Mens and T. Tourwé. *A Survey of Software Refactoring.* IEEE Transactions on Software Engineering, Vol. 30, No. 2, February 2004, p126–139.

[METZ01] P. Metz, J. O'Brien and W. Weber. *Against Use Case Interleaving.* IEEE Computer Society — UML 2001 Conference. LNCS 2185, p472–486, Springer Verlag. Toronto, 2001.

[MIKK97] E. Mikk, Y. Lakhnech, C. Petersohn and M. Siegel. *On Formal Semantics of Statecharts as Supported by STATEMATE.* In Second BCS-FACS Northern Formal Methods Workshop, 1997. Springer Verlag.

[MIL80] R. Milner. *A Calculus of Communicating Systems.* Number 92 in Lecture Notes in Computer Science. Springer Verlag, 1980.

[MIL99] G. Miller, J. McGregor and M. Major. *Capturing Framework Requirements*, Building Application Frameworks: Object-Oriented Foundations of Framework Design. M.E. Fayad, D.C. Schmidt, and R.E. Johnson (eds), p309–324, John Wiley & Sons, NY, 1999. ISBN: 0471248754.

[MSC00] *ITU-T. Recommendation Z.120, Message Sequence Charts.* Geneva, 1999.

[MSC92] *ITU-T. Recommendation Z.120, Message Sequence Charts.* Geneva, 1993.

[MSC96] *ITU-T. Recommendation Z.120, Message Sequence Charts.* Geneva, 1996.

[NUS00] B. Nuseibeh and S. Easterbrook. *Requirements Engineering: a Roadmap.* ICSE '00: Proceedings of the Conference on The Future of Software Engineering, p35–46. ACM Press, 2000. ISBN: 1-58113-253-0.

[OPD92] W. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois, 1992.

[ORAC] Oracle. http://www.oracle.com/tools/jdev_home.html.

[PARK81] D. M. R. Park. *Concurrency and Automata on Infinite Sequences.* Proceedings 5th GI Conference, number 104 in LNCS, p167–183. Springer Verlag, 1981.

[PET62] C. A. Petri. *Kommunikation Mit Automaten.* PhD thesis, Institut fuer Instrumentelle Mathematik, Bonn, 1962.

[PNU91] A. Pnueli and M. Shalev. *What is in a Step: On the Semantics of Statecharts.* In Proceedings Symposium on Theoretical Aspects of Computer Software, number 526 in Lecture Notes in Computer Science, p244–264. Springer Verlag, 1991.

[PRE92] R. Pressman. *Software Engineering: a Practitioner's Approach.* McGraw Hill, New York, 1992. ISBN: 0070508143.

[REG99] B. Regnell. *Requirements Engineering with Use Cases — A Basis for Software Development.* PhD thesis, Lund University, 1999.

[REN03] S. Ren, K. Rui and G. Butler. *Refactoring the Scenario Specification: a Message Sequence Chart Approach.* Object-Oriented Information Systems, 9th International Conference, OOIS 2003, Geneva, Switzerland, 2003, Proceedings. Lecture Notes in Computer Science 2817, Springer 2003, p294–298.

[REN04] S. Ren, G. Butler, K. Rui, J. Xu, W. Yu and R. Luo. *A Prototype Tool for Use Case Refactoring.* Proceedings of the 6th International Conference on Enterprise Information Systems, p173–178. Porto, Portugal, 2004.

[ROLE] RoleModel Software. *Drawlets Framework.*
http://www.rolemodelsoftware.com/drawlets/index.php.

[RUI03] K. Rui and G. Butler. *Refactoring Use Case Models: the Metamodel.* Proceedings of the Twenty-Sixth Australasian Computer Science Conference on Conference in Research and Practice in Information Technology, Vol. 16, p301–308. Adelaide, Australia, 2003.

[RUIR03] K. Rui, S. Ren and G. Butler. *Refactoring Use Case Models: a Case Study.* Proceedings of the 5th International Conference on Enterprise Information Systems, p239–244. Angers, France, 2003.

[SCO71] D. S. Scott and C. Strachey. *Towards a Mathematical Semantics for Computer Languages.* Proceedings Symposium Computers and Automata, p19–46. Polytechnic Institute of Brooklyn Press, 1971.

[SEL93] B. Henderson-Sellers and J. M. Edwards. *MOSES: an Overview.* In Proceedings of the Eleventh international Conference on Technology of Object-Oriented Languages and Systems, p561–571, 1993.

[SEL97] B. Henderson-Sellers, D. G. Firesmith and I. Graham. *OML Metamodel: Relationships and State Modeling*. Journal of Object Oriented Programming, 10(1), p47–51, March/April, 1997.

[SELL97] B. Henderson-Sellers and D. G. Firesmith. *Choosing between UML and OPEN.* American Programmer, 10(3), p15–23, 1997.

[SPI92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[STA] The Standish Group. http://www.tandishgroup.com.

[ST03] H. Störrle. *Semantics of UML 2.0 Activities.* http://www.pst.informatik.uni-muenchen.de/personen/stoerrle/V/AD1-CtrlFlow_VLHCC04.pdf.

[STO03] H. Störrle. *Semantics of Interactions in UML 2.0.* 2003 IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003), October 2003, Auckland, New Zealand. IEEE Computer Society 2003, ISBN: 0-7803-8225-0.

[STO99] H. Störrle. *A Petri-net Semantics for Sequence Diagrams.* Formale Beschreibungstechniken für verteilte Systeme, 9. GI/ITG-Fachgespräch, München, 1999. Herbert Utz Verlag 1999, ISBN: 3-89675-918-3.

[STV01] P. Stevens. *On Use Cases and Their Relationships in the Unified Modelling Language.* Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering, Genova, Italy, 2001. Springer LNCS 2029, ISBN: 3-540-41863-6, p140–155.

[SUNY01] G. Sunyé, D. Pollet, Y. L. Traon and J.-M. Jézéquel. *Refactoring UML Models.* In Proceedings of UML 2001 Conference, Springer Verlag LNCS 2185, Toronto, Canada, October 2001, p138–148.

[SUT98] A. G. Sutcliffe, N. A. M. Maiden, S. Minocha and D. Manuel. *Supporting Scenario-Based Requirements Engineering.* IEEE Transactions on Software Engineering, vol. 24, issue 12, (December), p1072-1088, 1998.

[TOK99] L. Tokuda. *Evolving Object-Oriented Designs with Refactorings.* PhD thesis, University of Texas at Austin, 1999.

[UML20] OMG. *Unified Modeling Language: Superstructure 2.0.* 2003, OMG.

[UML99] J. Rumbaugh, I. Jacobson and G. Booch. *The Unified Modeling Language Reference Manual.* Reading, MA: Addison Wesley, 1999.

[USE93] A. C. Uselton and S. A. Smolka. *State Refinement in Process Algebra.* In Proceedings of the North American Process Algebra Workshop, Ithaca, New York, August 1993.

[USE94] A. C. Uselton and S. A. Smolka. *A Compositional Semantics for Statecharts using Labeled Transition Systems.* In CONCUR '94, vol. 836 of LNCS, p2–17, 1994, Springer Verlag.

[USEL94] A. C. Uselton and S. A. Smolka. *A Process Algebraic Semantics for Statecharts via State Refinement.* In Proceedings of IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET), p267–286, June 1994.

[VAN01] A. Van Lamsweerde. *Goal-Oriented Requirements Engineering: A Guided Tour.* Invited Paper for RE'01, 5th IEEE International Symposium on Requirements Engineering, p249–263. Toronto, 2001.

[WIE96] R. J. Wieringa. *Requirements Engineering: Frameworks for Understanding.* John Wiley & Sons Inc., 1996.

[XU04] J. Xu. *On Refactoring of Use Case Models.* Master thesis, Concordia University, 2004.

[XU06] L. Xu. *Cascaded Refactoring for Framework Development and Evolution.* PhD thesis, Concordia University, 2006.

[YU04] W. Yu. *Refactoring Use Case Models on Episodes.* Master thesis, Concordia University, 2004.

APPENDIX: SPECIFICATION OF USE CASE REFACTORINGS

This section presents the specification of use case refactorings. These refactorings are grouped into three levels: environment level, structure level and event level. There are several categories at each level. Each refactoring is described in detail according to the template defined in Section 3.4.3. The refactoring is defined using notations and functions described in Section 3.4.2.

A. Refactorings at the Environment Level

A.1 Create an Environmental Entity

A.1.1 Create_Empty_Usecase

Description: create a new use case without any defined episode.

Arguments: string newUsecaseName.

Preconditions:

(a). The name of the new use case does not clash with an already existing use case within the model.

$$\forall usecase \in Model.usecases, usecase.name \neq newUsecaseName.$$

Postconditions:

(a). $\exists$ u $\in$ Model′.usecases $\wedge$ u $\notin$ Model.usecases $\wedge$ u.name = newUsecaseName
$\wedge$ u.episodes = {} $\wedge$ u.actors = {} $\wedge$ u.usecases = {}

Verification: The newly created use case does not have association relationship with any actor. It does not have any defined episode. It does not interact with any actor. The behavior of the use case model is not affected when this use case is added. The precondition ensures distinct environmental entity name (invariant two in Section 3.3).

A.1.2 Create_Empty_Actor

Description: create a new actor without any goal. It does not have any reference to other actors, users, and use cases.

Arguments: string newActorName.

Preconditions:

(a). The name of the new actor does not clash with an already existing actor within the model.

$$\forall actor \in Model.actors, actor.name \neq newActorName.$$

Postconditions:

(a). $\exists$ a $\in$ Model$'$.actors $\wedge$ a $\notin$ Model.actors $\wedge$ a.name = newActorName $\wedge$ a.usecases = {} $\wedge$ a.goals = {} $\wedge$ a.users = {} $\wedge$ a.actors = {}

Verification: When the new actor is created, it does not interact with any use case. It is isolated from other actors, goals and users. Therefore, the behavior of the use case model does not change when this new actor is added. The precondition ensures distinct environmental entity name (invariant two in Section 3.3).

A.1.3 Create_Empty_User

Description: create a new user without any task for a given actor.

Arguments: actor A, string newUserName.

Preconditions:

(a). The name of the new user does not clash with an already existing user within the model.

$$\forall user \in Model.users, user.name \neq newUserName.$$

Postconditions:

(a). $\exists$ u $\in$ A'.users $\wedge$ u $\notin$ A.users $\wedge$ u.name = newUserName
$\wedge$ u.usecases = {} $\wedge$ u.tasks = {} $\wedge$ u.actors = $\{A\}$

Verification: Creating a new user for an actor does not change interactions between the actor and related use cases. It is isolated from the task. Therefore, it does not change the behavior of the use case model. The precondition ensures distinct environmental entity name (invariant two in Section 3.3).

A.1.4 Create_Empty_Service

Description: create a new service in the use case model.

Arguments: string newServiceName.

Preconditions:

(a). The name of the new service does not clash with an already existing service within the model.

$$\forall service \in Model.services, service.name \neq newServiceName.$$

Postconditions:

(a). $\exists$ s $\in$ Model′.services $\wedge$ s $\notin$ Model.services $\wedge$ s.name = newServiceName
$\wedge$ s.usecases = {} $\wedge$ u.goals = {} $\wedge$ u.tasks = {}

Verification: A service is a package of functional entities (features) offered to the users in order to satisfy one or more goals that the users have. Since it is empty, it is isolated from use cases, goals and tasks in the use case model. Adding an empty service does not change the behavior of the use case model. The precondition ensures distinct environmental entity name (invariant two in Section 3.3).

A.1.5 Create_Empty_Goal

Description: create a new goal in the use case model.

Arguments: string newGoalName.

Preconditions:

(a). The name of the new goal does not clash with an already existing goal within the model.

$$\forall goal \in Model.goals, goal.name \neq newGoalName.$$

Postconditions:

(a). $\exists$ g $\in$ Model′.goals $\wedge$ g $\notin$ Model.goals $\wedge$ g.name = newGoalName $\wedge$ g.actors = {}
$\wedge$ g.goals = {}

Verification: The new goal is isolated from existing goals and actors in the use case model. It does not affect any interaction between actors and use cases. Therefore, this refactoring does not change the behavior of the use case model. The precondition ensures distinct environmental entity name (invariant two in Section 3.3).

A.1.6 Create_Empty_Task

Description: create a new task in the use case model.

Arguments: string newTaskName.

Preconditions:

(a). The name of the new task does not clash with an already existing task within the model.

$$\forall task \in Model.tasks, task.name \neq newTaskName.$$

Postconditions:

(a). $\exists$ t $\in$ Model′.tasks $\wedge$ t $\notin$ Model.tasks $\wedge$ t.name = newTaskName $\wedge$ t.users = {}
$\wedge$ t.tasks = {}

Verification: The new task is isolated from existing tasks and users in the use case model. It does not affect any interaction between actors and use cases. Therefore, this refactoring does not change the behavior of the use case model. The precondition ensures distinct environmental entity name (invariant two in Section 3.3).

A.2 Delete an Environmental Entity

A.2.1 Delete_Unreferenced_Usecase

Description: delete an unreferenced use case from the use case model.

Arguments: use case *U*.

Preconditions:

(a). The use case *U* is isolated from other use cases and actors.

$$U \in Model.usecases \wedge U.usecases = \{\} \wedge U.actors = \{\}$$

Postconditions:

(a). The use case *U* is deleted from the use case model.

$$U \notin Model'.usecases$$

(b). The deletion is cascaded to the structure level. After the use case U is deleted from the use case model, some episodes may be unreferenced. These episodes are deleted by applying refactoring *Delete_Unreferenced_Episode*, which cascades the deletion to the event level.

$$(\forall e \in U.episodes \wedge e.usecases = \{\}) \Rightarrow DeleteUnreferencedEpisode(e).$$

Verification: Since the use case U is isolated from other use cases and actors, it does not affect interactions between actors and use cases. Deleting it does not change the behavior of the use case model.

A.2.2 Delete_Unreferenced_Actor

Description: delete an unreferenced actor from the use case model.

Arguments: actor A.

Preconditions:

(a). The actor A is isolated from other actors and use cases.

$$A \in Model.actors \wedge A.usecases = \{\} \wedge A.actors = \{\}$$

Postconditions:

(a). The actor A is deleted from the use case model.

$$A \notin Model'.actors$$

(b). After the actor A is deleted from the use case model, some users may be unreferenced. These users are deleted from the use case model by *Delete_Unreferenced_User* refactoring.

$$(\forall u \in A.users \wedge u.actors = \{\}) \Rightarrow DeleteUnreferencedUser(u).$$

(c). After the actor A is deleted from the use case model, some goals may be unreferenced. These goals are deleted from the use case model by *Delete_Unreferenced_goal* refactoring.

$$(\forall g \in A.goals \wedge g.actors = \{\} \wedge g.goals = \{\}) \Rightarrow DeleteUnreferencedGoal(g).$$

Verification: Since the actor A is isolated from other actors and use cases, it does not participate in interactions between actors and use cases. Therefore, deleting it does not change the behavior of the use case model.

A.2.3 Delete_Unreferenced_User

Description: delete an unreferenced user from the use case model.

Arguments: user U.

Preconditions:

(a). The user U is not an instance of any actor in the use case model.

$$U \in Model.users \wedge U.actors = \{\}$$

Postconditions:

(a). The user U is deleted from the use case model.

$$U \notin Model'.users$$

(b). After the user U is deleted from the use case model, some tasks may become isolated from any user. These tasks are deleted from the use case model by *Delete_Unreferenced_Task* refactoring.

$$(\forall t \in U.tasks \wedge t.users = \{\} \wedge t.tasks = \{\}) \Rightarrow DeleteUnreferencedTask(t)$$

Verification: Since the user U is not an instance of any actor in the use case model, it does not affect interactions between actors and use cases. Deleting it does not change the behavior of the use case model.

A.2.4 Delete_Unreferenced_Service

Description: delete an unreferenced service from the use case model.

Arguments: service S.

Preconditions:

(a). There is no use case to describe the service S.

$$S \in Model.services \wedge S.usecases = \{\}$$

Postconditions:

(a). The service *S* is deleted from the use case model.

$$S \notin Model'.services$$

Verification: A service is a package of functional entities (features) offered to the users in order to satisfy one or more goals that the users have. Since there is no use case to describe the service *S*, it does not offer any behavior to the users. Deleting it does not change the behavior of the use case model.

A.2.5 Delete_Unreferenced_Goal

Description: delete an unreferenced goal from the use case model.

Arguments: goal *G*.

Preconditions:

(a). The goal *G* is not referenced by any other goal or actor.

$$G \in Model.goals \wedge G.goals = \{\} \wedge G.actors = \{\}$$

Postconditions:

(a). The goal *G* is deleted from the use case model.

$$G \notin Model'.goals$$

Verification: Since the goal *G* does not have any relationship with other goals and it does not belong to any actor, deleting it does not affect the consistency of the use case model.

A.2.6 Delete_Unreferenced_Task

Description: delete an unreferenced task from the use case model.

Arguments: task *T*.

Preconditions:

(a). The task *T* is not referenced by any other task or user.

$$T \in Model.tasks \wedge T.tasks = \{\} \wedge T.users = \{\}$$

Postconditions:

(a). The task T is deleted from the use case model.

$$T \notin Model'.tasks$$

Verification: Since the task T does not have any relationship with other tasks and it does not belong to any user, deleting it does not affect the consistency of the use case model.

A.3 Change an Environmental Entity

A.3.1 Change_Usecase_Name

Description: change the name of a use case within the use case model.

Arguments: use case U, string newUsecaseName.

Preconditions:

(a). $U \in Model.usecases$

(b). The new use case name does not clash with an already existing use case within the use case model.

$$\forall usecase \in Model.usecases, usecase.name \neq newUsecaseName.$$

Postconditions:

(a). U.name = newUsecaseName

Verification: Precondition (b) ensures distinct environmental entity name (invariant two in Section 3.3). Changing the name of a use case does not change the behavior of the use case model.

A.3.2 Change_Actor_Name

Description: change the name of an actor within the use case model.

Arguments: actor A, string newActorName.

Preconditions:

(a). $A \in Model.actors$

(b). The new actor name does not clash with an already existing actor within the use case model.

$$\forall actor \in Model.actors, actor.name \neq newActorName.$$

Postconditions:

(a). A.name = newActorName

Verification: Precondition (b) ensures distinct environmental entity name (invariant two in Section 3.3). Changing the name of an actor does not change the behavior of the use case model.

A.3.3 Change_User_Name

Description: change the name of a user within the use case model.

Arguments: user U, string newUserName.

Preconditions:

(a). $U \in Model.users$

(b). The new user name does not clash with an already existing user within the use case model.

$$\forall user \in Model.users, user.name \neq newUserName.$$

Postconditions:

(a). U.name = newUserName

Verification: Precondition (b) ensures distinct environmental entity name (invariant two in Section 3.3). Changing the name of a user does not change the behavior of the use case model.

A.3.4 Change_Service_Name

Description: change the name of a service within the use case model.

Arguments: service S, string newServiceName.

Preconditions:

(a). $S \in Model.services$

(b). The new service name does not clash with an already existing service within the use case model.

$$\forall service \in Model.services, service.name \neq newServiceName.$$

Postconditions:

(a). S.name = newServiceName

Verification: Precondition (b) ensures distinct environmental entity name (invariant two in Section 3.3). Changing the name of a service does not change the behavior of the use case model.

A.3.5 Change_Goal_Name

Description: change the name of a goal within the use case model.

Arguments: goal G, string newGoalName.

Preconditions:

(a). $G \in Model.goals$

(b). The new goal name does not clash with an already existing goal within the use case model.

$$\forall goal \in Model.goals, goal.name \neq newGoalName.$$

Postconditions:

(a). G.name = newGoalName

Verification: Precondition (b) ensures distinct environmental entity name (invariant two in Section 3.3). Changing the name of a goal does not change the behavior of the use case model.

A.3.6 Change_Task_Name

Description: change the name of a task within the use case model.

Arguments: task T, string newTaskName.

Preconditions:

(a). $T \in Model.tasks$

(b). The new task name does not clash with an already existing task within the use case model.

$$\forall task \in Model.tasks, task.name \neq newTaskName.$$

Postconditions:

(a). *T*.name = newTaskName

Verification: Precondition (b) ensures distinct environmental entity name (invariant two in Section 3.3). Changing the name of a task does not change the behavior of the use case model.

A.4 Compose an Environmental Entity

A.4.1 Change_Parent_Usecase

Description: given a use case U_1, change its parent use case to use case U_2.

Arguments: use case U_1, use case U_2.

Preconditions:

(a). The use case U_1 is not the parent of any use case.

$$U_1 \in Model.usecases \wedge U_2 \in Model.usecases \wedge U_2 \notin U_1.usecases \wedge$$
$$(\nexists u \in Model.usecases \wedge u.parentUsecase = U_1)$$

(b). All episodes currently inherited in the use case U_1 will be identically inherited from the use case U_2.

$$U_1.inheritedEpisodes \subseteq U_2.episodes$$

(c). There does not exist such an episode E that exists in the use case U_2 but does not exist in the use case U_1 or its parent use cases.

$$\forall E \in U_2.episodes, E \in (U_1.episodes \cup U_1.inheritedEpisodes).$$

Postconditions:

(a). The use case U_2 becomes the parent of the use case U_1.

$$U_1.parentUsecase = U_2$$

Verification: Precondition (a) ensures that only the behavior of the use case U_1 may be affected. There is no need to analyse other use cases. Precondition (b) ensures that the use case U_2 has all the behavior in current parent use cases of the use case U_1. Precondition (c) ensures that the use case U_1 does not inherit any other behavior than the behavior in the use case U_1 and the behavior inherited from its current parent use cases. Therefore, the behavior of the use case U_1 is not changed after the refactoring. In other words, this refactoring is a behavior preserving transformation.

A.4.2 Change_Parent_Actor

Description: given an actor A_1, change its parent actor to actor A_2.

Arguments: actor A_1, actor A_2.

Preconditions:

(a). $A_1 \in Model.actors \wedge A_2 \in Model.actors \wedge A_2.actors = \{\}$

(b). The actor A_1 has at most one parent actor. If the actor A_1 has the parent actor, this parent actor has the same set of goals with the actor A_2. All use cases used by this parent actor are also used by the actor A_2.

$$\begin{aligned}&\nexists A_1.parentActor \vee \\ &(\exists a = A_1.parentActor \wedge (a.goals \subseteq A_2.goals) \\ &\wedge (a.usecases \subseteq A_2.usecases) \wedge \nexists a.parentActor)\end{aligned}$$

(c). There does not exist such a use case U that is used by the actor A_2 but not used by the actor A_1 or the parent actor of A_1.

$$\forall U \in A_2.usecases, U \in (A_1.usecases \cup A_1.inheritedUsecases).$$

Postconditions:

(a). The actor A_2 becomes the parent of the actor A_1.

$$A_1.parentActor = A_2$$

Verification: This refactoring changes the parent of the actor A_1. Precondition (a) isolates the actor A_2 from other actors. So A_2 does not have any relationship with the actor A_1. Precondition (b) ensures that the actor A_2 uses all use cases that are used by the parent of the actor A_1. It also ensures that references to the goal model are the same after the refactoring since the actor A_2 has the same set of goals with the actor A_1. Precondition (c) ensures that the actor A_2 does not participate in any other use case than those used by the actor A_1 before the refactoring.

This refactoring preserves the interaction between actors and use cases. It ensures the same reference to the goal model. Therefore, it is a behavior preserving transformation.

A.4.3 Usecase_Generalization_Merger

Description: When there is a generalization relationship between two use cases and the parent use case is an abstract use case, this refactoring can be used to merge the parent use case into its child use case. After this refactoring, the parent use case and the generalization relationship between two use cases are removed. This refactoring helps manage the use case granularity and maintain the appropriate abstraction level of the use case.

Arguments: use case U_1, its parent use case U_2.

Preconditions:

(a). There is a generalization relationship between use cases U_1 and U_2. The use case U_2 is an abstract use case. It is the parent use case of U_1. The use case U_1 is a concrete use case.

$$\begin{aligned} &U_1 \in Model.usecases \wedge U_2 \in Model.usecases \\ &\wedge U_1.parentUsecase = U_2 \wedge U_1.type = concrete \\ &\wedge U_2.type = abstract \end{aligned}$$

(b). The use case U_2 is not referenced by any other use case than U_1.

$$U_2.usecases = \{U_1\}$$

Postconditions:

(a). The use case U_2 is merged into the use case U_1 by replacing the episode placeholder in U_1 with the corresponding episode.

$$\forall e \in U_2.episodes, SubstitutePlaceholderWithEpisode(U_1, Placeholder\,[e]).$$

(b). The use case U_1 takes over the association relationship between the use case U_2 and related actors. The generalization relationship between use cases U_1 and U_2 is deleted. The use case U_2 is deleted.

$$(\forall a \in U_2.actors, a \in U_1.actors) \land$$
$$U_2 \notin Model'.usecases \land (\nexists U \in Model'.usecases, U = U_1.parentUsecase)$$

Verification: The semantics of the generalization relationship between use cases is discussed in Section 3.2.2. Since the use case U_2 is an abstract use case, the use case U_2 can be merged into the use case U_1 based on episode placeholders within U_1. The interaction between the use case U_2 and related actors is not changed after the refactoring. This refactoring is a behavior preserving transformation.

Precondition (a) specifies the nature of use cases U_1 and U_2. Precondition (b) isolates the use case U_2 from other use cases than U_1. This simplifies the refactoring definition.

Postcondition (a) ensures that the use case U_2 is merged into the use case U_1 properly. Postcondition (b) is obvious.

A.4.4 Usecase_Inclusion_Merger

Description: This refactoring merges the included use case into its base use case when there is an inclusion relationship between these two use cases. After this refactoring, the inclusion relationship is removed. The episode tree of the included use case is inserted into the base use case at the inclusion point. This refactoring helps manage the use case granularity and maintain the appropriate abstraction level of the use case.

Arguments: the base use case U_1, the included use case U_2.

Preconditions:

(a). There is an inclusion relationship between use cases U_1 and U_2. The use case U_1 includes the use case U_2. The use case U_1 is a concrete use case. The use case U_2 is an abstract use case. It is not referenced by any other use case than U_1.

$$U_1 \in Model.usecases \land U_2 \in Model.usecases$$
$$\land\ Inclusion\ [U_1, U_2] \land U_1.type = concrete$$
$$\land\ U_2.type = abstract \land U_2.usecases = \{U_1\}$$

Postconditions:

(a). The use case U_2 is merged into the use case U_1 by inserting the episode tree of the use case U_2 at the inclusion point. Let e be the pseudo-episode indicating the inclusion point of this inclusion relationship.

$$SubstituteEpisodeTreeInUsecase(U_1, T[e], U_2.episodeTree)$$

(b). The inclusion relationship between use cases U_1 and U_2 is deleted. The use case U_2 is deleted.

$$U_2 \notin Model'.usecases$$

Verification: The semantics of the inclusion relationship between use cases is analysed in Section 3.2.2. Merging the included use case into its base use case does not change the behavior of the use case model.

Precondition (a) ensures that the use case U_2 can be removed after the refactoring because it is an abstract use case and not referenced by any other use case.

Postcondition (a) ensures that the use case U_2 is merged into the use case U_1 properly. Postcondition (b) is obvious.

A.4.5 Usecase_Extension_Merger

Description: This refactoring merges the extending use case into its base use case when there is an extension relationship between these two use cases. After this refactoring, the extension relationship is removed. The episode tree of the extending use case is inserted into the base use case at the extension point. This refactoring helps manage the use case granularity and maintain the appropriate abstraction level of the use case.

Arguments: the base use case U_1, the extending use case U_2.

Preconditions:

(a). There is an extension relationship between use cases U_1 and U_2. The use case U_2 extends the use case U_1. The use case U_1 is a concrete use case. The use case U_2 is an abstract use case. It is not referenced by any other use case than U_1.

$$\begin{aligned}&U_1 \in Model.usecases \wedge U_2 \in Model.usecases\\ &\wedge Extension\ [U_1, U_2] \wedge U_1.type = concrete\\ &\wedge U_2.type = abstract \wedge U_2.usecases = \{U_1\}\end{aligned}$$

Postconditions:

(a). The use case U_2 is merged into the use case U_1 by inserting the episode tree of the use case U_2 at the extension point. Let e be the pseudo-episode indicating the extension point of this extension relationship.

$$SubstituteEpisodeTreeInUsecase(U_1, T[e], U_2.episodeTree)$$

(b). The extension relationship between use cases U_1 and U_2 is deleted. The use case U_2 is deleted.

$$U_2 \notin Model'.usecases$$

Verification: The semantics of the extension relationship between use cases is analysed in Section 3.2.2. Merging the extending use case into its base use case does not change the behavior of the use case model.

Precondition (a) ensures that the use case U_2 can be removed after the refactoring because it is an abstract use case and not referenced by any other use case.

Postcondition (a) ensures that the use case U_2 is merged into the use case U_1 properly. Postcondition (b) is obvious.

A.4.6 Usecase_Precedence_Merger

Description: This refactoring merges two use cases into one use case when there is a precedence relationship between these two use cases. After this refactoring, the precedence relationship is removed. The episode tree of the preceding use case is adjusted by adding the episode tree of the other. This refactoring helps manage the use case granularity and maintain the appropriate abstraction level of the use case.

Arguments: the preceding use case U_1, use case U_2.

Preconditions:

(a). There is a precedence relationship between use cases U_1 and U_2. The use case U_1 precedes to the use case U_2. The use case U_1 is a concrete use case. The use case U_2 is an abstract use case. It is not referenced by any other use case than U_1.

$$\begin{aligned} & U_1 \in Model.usecases \wedge U_2 \in Model.usecases \\ & \wedge\ Precedence\ [U_1, U_2] \wedge U_1.type = concrete \\ & \wedge\ U_2.type = abstract \wedge U_2.usecases = \{U_1\} \end{aligned}$$

Postconditions:

(a). The use case U_2 is merged into U_1 by adjusting the episode tree of the use case U_1. A *sequence* composite episode is created as the root node of the new episode tree of the use case U_1. The original episode tree of the use case U_1 becomes its first sub episode tree. The episode tree of the use case U_2 becomes its second sub episode tree.

$$\begin{aligned} & \exists E \notin Model.episodes \wedge E \in Model'.episodes \\ & \wedge\ U_1'.episodeTree = T[E] \wedge T[E] = T[U_1.rootEpisode \cdot U_2.rootEpisode] \end{aligned}$$

(b). The precedence relationship between use cases U_1 and U_2 is deleted. The use case U_2 is deleted.

$$U_2 \notin Model'.usecases$$

Verification: The semantics of the precedence relationship between use cases is analysed in Section 3.2.2. Before the refactoring, the process algebra term for use cases U_1 and U_2 is $P_{U_1} \cdot P_{U_2}$. After the refactoring, the use case U_2 is merged into the use case U_1, $P_{U_2'} = P_{U_1.rootEpisode} \cdot P_{U_2.rootEpisode}$, which equals to $P_{U_1} \cdot P_{U_2}$. Therefore, this refactoring is a behavior preserving transformation.

Precondition (a) ensures that the use case U_2 can be removed after the refactoring because it is an abstract use case and not referenced by any other use case.

Postcondition (a) ensures that the use case U_2 is merged into the use case U_1 properly by adjusting the episode tree. Postcondition (b) is obvious.

A.4.7 Usecase_Equivalence_Generation

Description: When two use cases are equivalent in behavior, two refactorings can be applied. One is the *Usecase_Equivalence_Generation* refactoring. The other is the *Reuse_Usecase* refactoring. The

Usecase_Equivalence_Generation refactoring establishes an equivalence relationship between two use cases if both have the equivalent episode model. This refactoring helps specify the use case relationship explicitly so that the use case model is more understandable.

Arguments: use case U_1, use case U_2.

Preconditions:

(a). The episode model of the use case U_1 is equivalent to the episode model of the use case U_2.

$$P_{U_1} = P_{U_2}$$

Postconditions:

(a). An equivalence relationship is established between use cases U_1 and U_2.

$$\exists Equivalence\ [U_1, U_2]$$

Verification: If two use cases are equivalent in behavior, it means that both use cases have the equivalent episode model. The episode model can be specified by process algebra. If both episode models can be specified using the same process algebra term, they are equivalent. Both episode trees are not necessarily identical. Two different episode trees can have the same semantics. For example, the episode trees $T[e_1 \cdot (e_2 + e_3)]$ and $T[e_1 \cdot e_2 + e_1 \cdot e_3]$ have different tree structures, but the semantics of the episode model is the same: $p_{e_1} \cdot (p_{e_2} + p_{e_3}) = p_{e_1} \cdot p_{e_2} + p_{e_1} \cdot p_{e_3}$.

The semantics of the equivalence relationship is analysed in Section 3.2.2. Since use cases U_1 and U_2 have the same semantics, specifying the equivalence relationship explicitly between them does not change the behavior of the use case model.

A.4.8 Reuse_Usecase

Description: When two use cases are equivalent in behavior, two refactorings can be applied. One is the *Usecase_Equivalence_Generation* refactoring. The other is the *Reuse_Usecase* refactoring. The *Reuse_Usecase* refactoring keeps only one of these two use cases and removes the other. The kept use case takes over all references on the removed use case. This refactoring helps improve the use case reusability by removing the redundant use case.

Arguments: use case U_1, use case U_2.

Preconditions:

(a). The episode model of the use case U_1 is equivalent to the episode model of the use case U_2.

$$P_{U_1} = P_{U_2}$$

Postconditions:

(a). The use case U_1 takes over all references to the use case U_2.

$$\begin{aligned}
&(\forall U \in U_2.usecases \wedge U \neq U_1, U \in U_1.usecases) \wedge \\
&(\forall A \in U_2.actors, A \in U_1.actors) \wedge \\
&(\forall u \in U_2.users, u \in U_1.users)
\end{aligned}$$

(b). The use case U_2 is deleted from the use case model.

$$U_2 \notin Model'.usecases$$

Verification: If two use cases are equivalent in behavior, it means that both use cases have the equivalent episode model. The episode model can be specified by process algebra. If both episode models can be specified using the same process algebra term, they are equivalent. Both episode trees are not necessarily identical. Two different episode trees can have the same semantics. For example, the episode trees $T[e_1 \cdot (e_2 + e_3)]$ and $T[e_1 \cdot e_2 + e_1 \cdot e_3]$ have different tree structures, but the semantics of the episode model is the same: $p_{e_1} \cdot (p_{e_2} + p_{e_3}) = p_{e_1} \cdot p_{e_2} + p_{e_1} \cdot p_{e_3}$.

Since use cases U_1 and U_2 are equivalent, removing the use case U_2 and keeping only the use case U_1 does not change the behavior of the use case model. Precondition (a) specifies that both use cases have the equivalent episode model using the process algebra term. Postcondition (a) ensures that the use case model is consistent after the refactoring by updating related references to the use case U_2. Postcondition (b) is obvious.

A.4.9 Merge_Usecases

Description: When two use cases are independent to each other and both use cases are used by the same set of actors, one use case can be merged into the other without affecting the behavior of the use case model. This refactoring keeps one of these two use cases and removes the other. This refactoring helps manage the use case granularity and avoid fragment use cases.

Arguments: use case U_1, use case U_2.

Preconditions:

(a). Use cases U_1 and U_2 are concrete use cases. They are not referenced by any use case.

$$
\begin{aligned}
& U_1 \in Model.usecases \land U_2 \in Model.usecases \\
& \land U_1.type = concrete \land U_2.type = concrete \\
& \land U_1.usecases = \{\} \land U_2.usecases = \{\}
\end{aligned}
$$

(b). Use cases U_1 and U_2 are used by the same set of actors.

$$U_1.actors = U_2.actors$$

Postconditions:

(a). The use case U_2 is merged into the use case U_1 by adjusting the episode tree of the use case U_1. An *alternation* composite episode is created as the root node of the new episode tree of the use case U_1. The original episode tree of U_1 and the episode tree of U_2 become its direct sub episode trees.

$$
\begin{aligned}
& \exists E \notin Model.episodes \land E \in Model'.episodes \\
& \land U_1'.episodeTree = T[E] \land T[E] = T[U_1.rootEpisode + U_2.rootEpisode]
\end{aligned}
$$

(b). The use case U_2 is deleted.

$$U_2 \notin Model'.usecases$$

Verification: As described in Section 3.2.2, two concrete use cases can be mapped into one use case which consists of the "or" of their respective episode models if both use cases are independent to each other. This refactoring is a behavior preserving transformation.

Precondition (a) specifies that use cases U_1 and U_2 are isolated from other use cases. This ensures that merging both use cases does not affect the behavior of other use cases. For example, if there is another use case U including the use case U_1, undesirable behavior will be introduced into the use case U as the result of merging the use case U_2 into U_1. Precondition (b) ensures that both use cases are used by the same set of actors. Otherwise, the merged use case is used by actors associated with both use cases U_1 and U_2. The interaction between the merged use case and related actors at the environmental level will become less clear. This reduces the understandability of the merged use case.

Postcondition (a) ensures that the use case U_2 is merged into the use case U_1 properly. Postcondition (b) is obvious.

A.4.10 Generalize_Usecases

Description: This refactoring generalizes use cases when two or more use cases have a common episode subtree. After the refactoring, an abstract use case is created as the parent of these use cases. The common episode subtree is moved into this new use case. This refactoring helps reduce the redundancy of the use case model and improve the reusability.

Arguments: a set of use cases $\{U_1, U_2, \cdots, U_n\}$, a common episode subtree t, string newUsecase-Name.

Preconditions:

(a). Use cases $\{U_1, U_2, \cdots, U_n\}$ are used by the same set of actors. They have a common episode subtree t.

$$\begin{aligned}\forall u \in \{U_1, U_2, \cdots, U_n\}, u \in Model.usecases \wedge \\ u.actors = U_1.actors \wedge \\ u.type = concrete \wedge \\ IsEpisodeSubTree(u.episodeTree, t)\end{aligned}$$

(b). There is no use case relationship among Use cases $\{U_1, U_2, \cdots, U_n\}$. These use cases are not referenced by any other use case.

$$\forall u \in \{U_1, U_2, \cdots, U_n\}, u.usecases = \{\}.$$

(c). The name of the new use case does not clash with an already existing use case within the use case model.

$$\forall usecase \in Model.usecases, usecase.name \neq newUsecaseName.$$

Postconditions:

(a). A new use case U is created with the name of newUsecaseName. The use case U is an abstract use case. It is the parent of use cases $\{U_1, U_2, \cdots, U_n\}$. The episode tree of the use case U is the episode subtree t.

$$\begin{aligned}
&\exists U \notin Model.usecases \wedge U \in Model'.usecases \\
&\wedge\ U.name = newUsecaseName \wedge U.type = abstract \\
&\wedge\ (\forall usecase \in \{U_1, U_2, \cdots, U_n\}, usecase.parentUsecase = U) \\
&\wedge\ U.episodeTree = t
\end{aligned}$$

(b). In use cases $\{U_1, U_2, \cdots, U_n\}$, episodes occurring in the episode subtree t are replaced by corresponding episode placeholders.

$$\begin{gathered}
\forall e \in t.episodes \wedge \forall u \in \{U_1, U_2, \cdots, U_n\}, \\
SubstituteEpisodeWithPlaceholder(u, e).
\end{gathered}$$

Verification: The generalization relationship means that the child use case contains all the attributes and behavior sequence defined in the parent use case, and participates in all relationships of the parent use case. The semantics of the generalization relationship is analysed in Section 3.2.2. This refactoring creates the parent use case U for use cases $\{U_1, U_2, \cdots, U_n\}$ and moves the common episode subtree t to U. The use case U is an abstract use case. It cannot be instantiated on its own. Since the common episode subtree t is inherited in use cases $\{U_1, U_2, \cdots, U_n\}$, the behavior of the use case model is preserved.

Precondition (a) ensures that use cases $\{U_1, U_2, \cdots, U_n\}$ have the common episode subtree t. Each use case has the same set of actors. According to the semantics of the generalization relationship, moving the episode subtree t to the parent use case does not change the behavior of these use cases. Precondition (b) ensures that use cases $\{U_1, U_2, \cdots, U_n\}$ are isolated. Refactoring these use cases does not affect the behavior of other use cases. Precondition (c) ensures distinct environmental entity name (invariant two in Section 3.3).

Postcondition (a) is obvious. It ensures that the new use case U is an abstract use case. Postcondition (b) ensures that episodes in the episode subtree t are replaced by corresponding placeholders properly in use cases $\{U_1, U_2, \cdots, U_n\}$.

A.4.11 Merge_Actors

Description: Two actors can be merged into one actor without affecting the behavior of the use case model. This refactoring keeps one actor and removes the other. The kept actor takes over all references to the removed actor. This refactoring helps manage actors.

Arguments: actor U_1, actor U_2.

Preconditions:

(a). Actors A_1 and A_2 are not referenced by any other actor, but the actor A_2 can be the parent actor of A_1.

$$\begin{aligned}
&A_1 \in \mathit{Model.actors} \wedge A_2 \in \mathit{Model.actors}\\
&\wedge A_1.\mathit{actors} \subseteq \{A_2\}\\
&\wedge A_2.\mathit{actors} \subseteq \{A_1\}\\
&\wedge (A_1.\mathit{actors} = \{\} \vee A_1.\mathit{parentActor} = A_2)
\end{aligned}$$

Postconditions:

(a). The actor A_2 is merged into the actor A_1: goals in the actor A_2 are moved into the actor A_1; users of A_2 become users of the actor A_1; use cases used by the actor A_2 are used by the actor A_1.

$$\begin{aligned}
&(\forall g \in A_2.\mathit{goals}, g \in A_1.\mathit{goals})\wedge\\
&(\forall u \in A_2.\mathit{users}, u \in A_1.\mathit{users})\wedge\\
&(\forall U \in A_2.\mathit{usecases}, U \in A_1.\mathit{usecases})
\end{aligned}$$

(b). The actor A_2 is deleted.

$$A_2 \notin \mathit{Model'.actors}$$

Verification: Merging two actors into one actor does not change interactions between actors and use cases. This refactoring is a behavior preserving transformation.

Precondition (a) ensures that actors A_1 and A_2 are isolated from other actors. Otherwise, merging the actor A_2 to A_1 may affect other actors.

According to the use case metamodel defined in Section 3.1, the actor can be referenced by the actor, use case, user or goal. Postcondition (a) ensures that references to the use case, user and goal in the actor A_2 are taken over by the actor A_1. This is very important for the behavior preservation. For example, if the association relationship between the actor A_2 and related use cases is not taken over by A_1, the interaction between the actor A_2 and related use cases is lost. Thus the behavior of the use case model is changed. Postcondition (b) is obvious. The actor A_2 is an unreferenced actor after the refactoring. It should be deleted from the use case model.

A.4.12 Generalize_Actors

Description: When two or more actors use a common set of use cases and have a common set of goals, this refactoring can be used to create a new actor A as the parent of these actors. The actor A uses the above common set of use cases. The common goals are moved into the actor A. This refactoring helps reduce the redundancy of the actor definition and improve the reusability of the use case model.

Arguments: a set of actors $\{A_1, A_2, \cdots, A_n\}$, a common set of use cases $\{U_1, U_2, \cdots, U_n\}$, a common set of goals $\{G_1, G_2, \cdots, G_n\}$, string newActorName.

Preconditions:

(a). Actors $\{A_1, A_2, \cdots, A_n\}$ use a common set of use cases $\{U_1, U_2, \cdots, U_n\}$ and have a common set of goals $\{G_1, G_2, \cdots, G_n\}$.

$$\begin{aligned}
&(\forall u \in \{U_1, U_2, \cdots, U_n\}, u \in \mathit{Model.usecases}) \land \\
&(\forall g \in \{G_1, G_2, \cdots, G_n\}, g \in \mathit{Model.goals}) \land \\
&(\forall a \in \{A_1, A_2, \cdots, A_n\}, a \in \mathit{Model.actors} \land \\
&\qquad\qquad (\{U_1, U_2, \cdots, U_n\} \subseteq \mathit{a.usecases}) \land \\
&\qquad\qquad (\{G_1, G_2, \cdots, G_n\} \subseteq \mathit{a.goals}))
\end{aligned}$$

(b). There is no actor relationship among actors $\{A_1, A_2, \cdots, A_n\}$. These actors are not referenced by any other actor.

$$\forall a \in \{A_1, A_2, \cdots, A_n\}, \mathit{a.actors} = \{\}.$$

(c). The name of the new actor does not clash with an already existing actor within the use case model.

$$\forall actor \in Model.actors, actor.name \neq newActorName.$$

Postconditions:

(a). A new actor A is created under the name of newActorName. The actor A is the parent of actors $\{A_1, A_2, \cdots, A_n\}$. It has the association relationship with use cases $\{U_1, U_2, \cdots, U_n\}$.

$$\begin{aligned}&\exists A \notin Model.actors \wedge A \in Model'.actors\\&\wedge A.name = newActorName\\&\wedge (\forall actor \in \{A_1, A_2, \cdots, A_n\}, actor.parentActor = A)\\&\wedge A.usecases = \{U_1, U_2, \cdots, U_n\}\end{aligned}$$

(b). Association relationships between use cases $\{U_1, U_2, \cdots, U_n\}$ and actors $\{A_1, A_2, \cdots, A_n\}$ are removed. Actors $\{A_1, A_2, \cdots, A_n\}$ inherit these association relationships from the parent actor A.

$$\begin{aligned}&\forall a \in \{A_1, A_2, \cdots, A_n\}, \nexists u \in \{U_1, U_2, \cdots, U_n\}\\&\qquad\qquad \wedge u \in a.usecases.\end{aligned}$$

(c). Common goals $\{G_1, G_2, \cdots, G_n\}$ are removed from actors $\{A_1, A_2, \cdots, A_n\}$ and moved to the new actor A.

$$\begin{aligned}&\forall g \in \{G_1, G_2, \cdots, G_n\}, g \in A.goals\\&\qquad\qquad \wedge (\nexists actor \in \{A_1, A_2, \cdots, A_n\}\\&\qquad\qquad \wedge g \in actor.goals).\end{aligned}$$

Verification: The generalization relationship between actors means that the child actor contains all the attributes and behavior sequences defined in the parent actor, and participates in all relationships of the parent actor. The new actor A represents common roles played by actors $\{A_1, A_2, \cdots, A_n\}$. It does not add any new interaction between actors and use cases. Actors $\{A_1, A_2, \cdots, A_n\}$ inherit association relationships between the actor A and use cases $\{U_1, U_2, \cdots, U_n\}$. All the interactions between actors and use cases are preserved. This refactoring is a behavior preserving transformation.

Precondition (a) is obvious. It represents common roles played by actors $\{A_1, A_2, \cdots, A_n\}$. Precondition (b) ensures that actors $\{A_1, A_2, \cdots, A_n\}$ are isolated from other actors so that the new actor A

does not affect other actors. Precondition (c) ensures distinct environmental entity name (invariant two in Section 3.3).

Postcondition (a) is obvious. Postcondition (b) adjusts association relationships between actors and use cases. Postcondition (c) moves common goals from actors $\{A_1, A_2, \cdots, A_n\}$ to the new actor A. This removes the redundancy and improves the understandability of the use case model.

A.5 Decompose an Environmental Entity

A.5.1 Split_Usecase

Description: This refactoring splits one use case into two use cases. An episode subtree in the original use case is moved into the new use case. This refactoring helps manage the use case granularity.

Arguments: use case U, the episode tree T of the use case U, the episode subtree t_1 for creating the new use case, string newUsecaseName.

Preconditions:

(a). The episode subtree t_1 is a subtree of T.

$$U \in Model.usecases \wedge U.episodeTree = T \wedge IsEpisodeSubTree(T, t_1)$$

(b). The use case U is not referenced by any other use case.

$$U.usecases = \{\}$$

(c). The root node of the episode tree T is an alternation composite episode. The episode tree T has two direct episode subtrees. The episode tree t_1 is one of its direct episode subtrees.

$$\begin{aligned} & U.rootEpisode.type = AlternationEpisode \\ & \wedge \exists e_1 \in U.episodes \wedge \exists e_2 \in U.episodes \\ & \wedge T[e_1] = t_1 \wedge T = T[e_1 + e_2] \end{aligned}$$

(d). The name of the new use case does not clash with an already existing use case within the use case model.

$$\forall usecase \in Model.usecases, usecase.name \neq newUsecaseName.$$

Postconditions:

(a). A new use case U_1 is created with the name of newUsecaseName. The episode tree of use case U_1 is the episode subtree t_1.

$$\exists U_1 \notin Model.usecases \wedge U_1 \in Model'.usecases \wedge U_1.episodeTree = t_1$$

(b). The new use case U_1 is used by all actors that have the association relationship with the use case U. There is no use case relationship between use cases U and U_1.

$$U_1.actors = U.actors \wedge U_1.usecases = \{\}$$

(c). In the use case U, the episode tree T is changed into T' by removing the episode subtree t_1 from T.

$$\begin{aligned}&\exists e_1 \in U.episodes \wedge \exists e_2 \in U.episodes\\&\wedge e_1 \notin U'.episodes \wedge e_2 \in U'.episodes\\&\wedge T[e_1] = t_1 \wedge T = T[e_1 + e_2]\\&\wedge T' = T[e_2] \wedge U'.episodeTree = T'\end{aligned}$$

Verification: Conversely to the *Merge_Usecases* refactoring, this refactoring splits one use case into two use cases. The new use case U_1 has no use case relationship with the split use case U'. Before the refactoring, $P_U = P_{e_1} + P_{e_2}$. After the refactoring, the process algebra term for the use case model containing use cases U_1 and U' is: $P_{U_1} + P_{U'} = P_{e_1} + P_{e_2}$. Therefore, the behavior of the use case model is preserved.

Precondition (a) is obvious. Precondition (b) ensures that the use case U has no use case relationship with other use cases so that splitting U does not affect the behavior of other use cases. Precondition (c) specifies the pattern of the episode tree of the use case U. The root episode of the use case U has to be an alternation composite episode. Otherwise, there may exist the use case relationship between the use case U' and the new use case U_1. Precondition (d) ensures distinct environmental entity name (invariant two in Section 3.3).

Postcondition (a) and (b) are obvious. Postcondition (c) ensures that the episode tree T is adjusted properly after the refactoring. The episode subtree t_1 has to be removed from the episode tree T.

A.5.2 Usecase_Generalization_Generation

Description: This refactoring splits one use case into two and generates a generalization relationship between these two use cases. After this refactoring, a new use case is created as the parent of the other use case. One of its episode subtrees is moved to the new use case. This refactoring helps manage the use case granularity. It can be seen as a special case of the *Generalize_Usecases* refactoring when the set of use cases contains only one use case.

Arguments: use case U, the episode tree T of the use case U, the episode subtree t_1 for creating the new use case, string newUsecaseName.

Preconditions:

(a). The episode tree t_1 is a subtree of T.

$$U \in Model.usecases \land U.episodeTree = T \land IsEpisodeSubTree(T, t_1)$$

(b). The use case U is not referenced by any other use case.

$$U.usecases = \{\}$$

(c). The name of the new use case does not clash with an already existing use case within the use case model.

$$\forall usecase \in Model.usecases, usecase.name \neq newUsecaseName.$$

Postconditions:

(a). A new use case U_1 is created with the name of newUsecaseName. The episode tree of the use case U_1 is the episode tree t_1. The use case U_1 is an abstract use case. It is the parent of the use case U.

$$\exists U_1 \notin Model.usecases \land U_1 \in Model'.usecases \land U_1.type = abstract$$
$$\land\ U_1.episodeTree = t_1 \land U.parentUsecase = U_1$$

(b). In the use case U, episodes occurring in the episode subtree t_1 are replaced by corresponding episode placeholders.

$$\forall e \in t_1.episodes, SubstituteEpisodeWithPlaceholder(U, e).$$

Verification: Conversely to the *Usecase_Generalization_Merger* refactoring, this refactoring splits one use case into two and generates a generalization relationship between these two use cases. The semantics of the generalization relationship is analysed in Section 3.2.2. The new use case U_1 is an abstract actor. It cannot be instantiated on its own. The use case U inherits the episode subtree t_1 from the use case U_1. The interaction between actors and the use case U is preserved. Therefore, this refactoring is a behavior preserving transformation.

Precondition (a) is obvious. Precondition (b) ensures that the use case U has no use case relationship with other use cases so that splitting U does not affect the behavior of other use cases. Precondition (c) ensures distinct environmental entity name (invariant two in Section 3.3).

Postcondition (a) is obvious. Postcondition (b) ensures that episodes in the episode subtree t_1 are replaced by corresponding episode placeholders properly.

A.5.3 Usecase_Inclusion_Generation

Description: This refactoring splits one use case into two and generates an inclusion relationship between these two use cases. When the episode model of a use case is too complicated, it is preferable to split the episode tree. After the refactoring, a new use case is created with an episode subtree of the original use case. The original use case includes the new use case. This refactoring helps manage the use case granularity, reduce the redundancy, and improve the understandability of the use case model.

Arguments: use case U, the episode tree T of the use case U, the episode subtree t_1 for creating the new use case, string newUsecaseName.

Preconditions:

(a). The episode subtree t_1 is a subtree of T.

$$U \in Model.usecases \wedge U.episodeTree = T \wedge IsEpisodeSubTree(T, t_1)$$

(b). The name of the new use case does not clash with an already existing use case within the use case model.

$$\forall usecase \in Model.usecases, usecase.name \neq newUsecaseName.$$

Postconditions:

(a). An abstract use case U_1 is created with the name of newUsecaseName. The episode tree of the new use case U_1 is the episode subtree t_1. The use case U includes the new use case U_1.

$$\exists U_1 \notin Model.usecases \wedge U_1 \in Model'.usecases$$
$$\wedge\ U_1.type = abstract \wedge U_1.episodeTree = t_1 \wedge Inclusion[U, U_1]$$

(b). The episode tree t_1 is substituted by a pseudo-episode in the episode tree T. The pseudo-episode indicates the inclusion point for the inclusion relationship between use cases U and U_1.

$$SubstituteEpisodeTreeInUsecase(U, t_1, T[Pseudo[U_1]])$$

Verification: Conversely to the *Usecase_Inclusion_Merger* refactoring, this refactoring splits one use case into two and generates an inclusion relationship between these two use cases. The semantics of the inclusion relationship is analysed in Section 3.2.2. The new use case U_1 is an abstract use case. It cannot be instantiated on its own. The use case U' includes the behavior of the new use case U_1 at the inclusion point. The interaction between the use case U and related actors is preserved. Therefore, this refactoring is a behavior preserving transformation.

Precondition (a) is obvious. Precondition (b) ensures distinct environmental entity name (invariant two in Section 3.3).

Postcondition (a) ensures that the new use case U_1 is an abstract use case. Therefore, the behavior of the use case model with use cases U' and U_1 can be verified based on the interaction between the use case U' and related actors. Postcondition (b) ensures that the episode tree T is adjusted properly after the refactoring by replacing the episode subtree t_1 with the inclusion point.

A.5.4 Usecase_Extension_Generation

Description: This refactoring splits one use case into two and generates an extension relationship between these two use cases. When the episode subtree for an exceptional or alternative course is too complicated, it is preferable to split the episode tree of the use case. After the refactoring, a new use case is created with the episode subtree representing the exceptional or alternative course. The new use case extends the original use case. This refactoring helps manage the use case granularity, reduce the redundancy, and improve the understandability of the use case model.

Arguments: use case U, the episode tree T of the use case U, the episode subtree t_1 representing the exceptional or alternative course, string newUsecaseName.

Preconditions:

(a). The episode subtree t_1 is a subtree of T. It represents an exceptional or alternative course.

$$\begin{aligned}
&U \in Model.usecases \wedge U.episodeTree = T \wedge IsEpisodeSubTree(T, t_1)\\
&\wedge \exists e_0 \in U.episodes \wedge \exists e_1 \in U.episodes \wedge \exists e_2 \in U.episodes\\
&\wedge e_0.type = AlternationEpisode \wedge T[e_0] = T[e_1 + e_2] \wedge e_1 \neq e_2\\
&\wedge T[e_1] = t_1
\end{aligned}$$

(b). The name of the new use case does not clash with an already existing use case within the use case model.

$$\forall usecase \in Model.usecases, usecase.name \neq newUsecaseName.$$

Postconditions:

(a). An abstract use case U_1 is created with the name of newUsecaseName. The episode tree of the use case U_1 is the episode subtree t_1. The use case U_1 extends the use case U.

$$\begin{aligned}
&\exists U_1 \notin Model.usecases \wedge U_1 \in Model'.usecases\\
&\wedge U_1.type = abstract \wedge U_1.episodeTree = t_1 \wedge Extension[U, U_1]
\end{aligned}$$

(b). The episode tree t_1 is substituted by a pseudo-episode in the episode tree T. The pseudo-episode indicates the extension point for the extension relationship between use cases U and U_1.

$$SubstituteEpisodeTreeInUsecase(U, t_1, T[Pseudo[U_1]])$$

Verification: Conversely to the *Usecase_Extension_Merger* refactoring, this refactoring splits one use case into two and generates an extension relationship between these two use cases. The semantics of the extension relationship is analysed in Section 3.2.2. The new use case U_1 is an abstract use case. It cannot be instantiated on its own. The new use case U_1 extends the use case U at the extension point. The interaction between actors and the use case U is preserved. Therefore, this refactoring is a behavior preserving transformation.

Precondition (a) ensures that the episode subtree t_1 represents an exceptional or alternative course in the use case U so that t_1 can be moved to the extending use case U_1. Precondition (b) ensures distinct environmental entity name (invariant two in Section 3.3).

Postcondition (a) ensures that the new use case U_1 is an abstract use case. Therefore, the behavior of the use case model with use cases U' and U_1 can be verified based on the interaction between the use case U' and related actors. Postcondition (b) ensures that the episode tree T is adjusted properly after the refactoring by replacing the episode subtree t_1 with the extension point.

A.5.5 Usecase_Precedence_Generation

Description: This refactoring splits one use case into two and generates a precedence relationship between these two use cases. When an episode subtree is always performed at the end of the execution and the subtree is complicated, it is preferable to move this subtree to a new use case. A precedence relationship is established to specify the time order between the original use case and the new use case. This refactoring helps manage the use case granularity, reduce the redundancy, and improve the understandability of the use case model.

Arguments: use case U, the episode tree T of the use case U, the episode subtree t_1 for creating the new use case, string newUsecaseName.

Preconditions:

(a). The use case U is not referenced by any other use case.

$$U \in Model.usecases \wedge U.usecases = \{\}$$

(b). The episode subtree t_1 is a subtree of T. The episode subtree t_1 is always performed at the end.

$$\begin{aligned}&U.episodeTree = T \wedge IsEpisodeSubTree(T, t_1)\\&\wedge \exists e_0 \in U.episodes \wedge \exists e_1 \in U.episodes \wedge e_0 \neq e_1\\&\wedge T = T[e_0 \cdot e_1] \wedge t_1 = T[e_1]\end{aligned}$$

(c). The name of the new use case does not clash with an already existing use case within the use case model.

$$\forall usecase \in Model.usecases, usecase.name \neq newUsecaseName.$$

Postconditions:

(a). An abstract use case U_1 is created with the name of newUsecaseName. The episode tree of the use case U_1 is the episode subtree t_1. The use case U precedes to the use case U_1.

$$\exists U_1 \notin Model.usecases \land U_1 \in Model'.usecases$$
$$\land\, U_type = abstract \land U_1.episodeTree = t_1 \land Precedence[U, U_1]$$

(b). The episode subtree t_1 is removed from the episode tree T.

$$DeleteEpisodeTreeInUsecase(U, t_1)$$

Verification: Conversely to the *Usecase_Precedence_Merger* refactoring, this refactoring splits one use case into two and generates a precedence relationship between these two use cases. The semantics of the precedence relationship is analysed in Section 3.2.2. The new use case U_1 is an abstract use case. It cannot be instantiated on its own. The new use case U_1 is always performed after the use case U. The interaction between actors and the use case U is preserved. Therefore, this refactoring is a behavior preserving transformation.

Precondition (a) ensures that the use case U has no use case relationship with other use cases. Otherwise, splitting it may affect the behavior of other use cases. Precondition (b) ensures that the episode subtree t_1 is always performed at the end in the use case U. Otherwise, the precedence relationship cannot be created between use cases U and U_1. Precondition (c) ensures distinct environmental entity name (invariant two in Section 3.3).

Postcondition (a) ensures that the new use case U_1 is an abstract use case. Therefore, the behavior of the use case model with use cases U' and U_1 can be verified based on the interaction between the use case U' and related actors. Postcondition (b) ensures that the episode tree T is adjusted by removing the episode subtree t_1 after the refactoring.

A.5.6 Split_Actor

Description: This refactoring splits one actor into two actors. After the refactoring, a new actor is created. The new actor has a subset of goals that the original actor has. The new actor takes over some roles that the original actor plays. This refactoring helps manage the actor granularity. It improves the understandability and reusability of the use case model.

Arguments: actor A, a set of goals $\{G_1, G_2, \cdots, G_n\}$ for creating the new actor, string newActorName.

Preconditions:

(a). The set of goals $\{G_1, G_2, \cdots, G_n\}$ is a subset of goals of the actor A.

$$A \in Model.actors \wedge (\{G_1, G_2, \cdots, G_n\} \subset A.goals)$$

(b). The actor A interacts with one use case in the use case model. The actor A has only one user.

$$(\exists U \in Model.usecases, A.usecases = \{U\})$$
$$\wedge\ (\exists user \in Model.users, A.user = \{user\})$$

(c). The actor A has no actor relationship with any other actor.

$$\nexists A.actors = \{\}$$

(d). The name of the new actor does not clash with an already existing actor within the use case model.

$$\forall actor \in Model.actors, actor.name \neq newActorName.$$

Postconditions:

(a). A new actor A_1 is created with the name of newActorName. The actor A_1 has goals $\{G_1, G_2, \cdots, G_n\}$. The actor A_1 does not have the actor relationship with the actor A'. It interacts with the use case that the actor A interacts with.

$$A_1 \notin Model.actors \wedge A_1 \in Model'.actors$$
$$\wedge\ A_1.goals = \{G_1, G_2, \cdots, G_n\} \wedge A_1.actor = \{\}$$
$$\wedge\ A_1.usecases = A.usecases \wedge A_1.users = A.users$$

(b). The actor A' does not have any goal in goals $\{G_1, G_2, \cdots, G_n\}$.

$$\forall g \in \{G_1, G_2, \cdots, G_n\}, g \notin A'.goals.$$

Verification: This refactoring splits one actor into two actors. The new actor A_1 interacts with the same use case U that the actor A interacts with. The interaction between the actor A and the use case U is preserved by the interaction between the actor A' and the use case U and the interaction between the actor A_1 and the use case U. Thus this refactoring is a behavior preserving transformation.

Precondition (a) ensures that the set of goals $\{G_1, G_2, \cdots, G_n\}$ is a subset of goals of the actor A. So the new actor A_1 represents some of roles played by the actor A. Precondition (b) ensures that the actor A interacts with only one use case. The actor A has only one user. This simplifies the definition of this refactoring. Precondition (c) ensures that the use case A does not have the actor relationship with any other actor. Otherwise, splitting it may affect the behavior of other actors. Precondition (d) ensures distinct environmental entity name (invariant two in Section 3.3).

Postcondition (a) ensures that the new actor A_1 is created properly. Postcondition (b) ensures that the actor A is adjusted by removing goals $\{G_1, G_2, \cdots, G_n\}$.

A.5.7 Actor_Generalization_Generation

Description: This refactoring splits one actor into two actors and generates a generalization relationship between these two actors. After the refactoring, a new abstract actor is created. A set of goals representing the general behavior of the original actor is moved to the new actor. This refactoring helps improve the understandability and reusability of the use case model.

Arguments: actor A, goals $\{G_1, G_2, \cdots, G_n\}$ for creating the new actor, string newActorName.

Preconditions:

(a). Goals $\{G_1, G_2, \cdots, G_n\}$ represents the general behavior of the actor A. It is a subset of goals of the actor A.

$$A \in Model.actors \wedge (\{G_1, G_2, \cdots, G_n\} \subset A.goals)$$

(b). The actor A does not have the parent actor.

$$\forall actor \in Model.actors, actor \neq A.parentActor.$$

(c). The name of the new actor does not clash with an already existing actor within the use case model.

$$\forall actor \in Model.actors, actor.name \neq newActorName.$$

Postconditions:

(a). A new abstract actor A_1 is created with the name of newActorName. The actor A_1 is the parent of the actor A. The actor A_1 has goals $\{G_1, G_2, \cdots, G_n\}$. The actor A does not have any goal in goals $\{G_1, G_2, \cdots, G_n\}$.

$$\begin{aligned}
&A_1 \notin Model.actors \wedge A_1 \in Model'.actors \\
&\wedge A_1.type = abstract \wedge A'.parentActor = A_1 \\
&\wedge A_1.goals = \{G_1, G_2, \cdots, G_n\} \\
&\wedge (\forall g \in \{G_1, G_2, \cdots, G_n\}, g \notin A'.goals)
\end{aligned}$$

Verification: This refactoring splits one actor into two and generates a generalization relationship between these two actors. The new actor A_1 is an abstract actor. The interaction between the actor A and use cases is preserved. Since the actor A' inherits goals $\{G_1, G_2, \cdots, G_n\}$ from the actor A_1, the actor A' has the same set of goals with the actor A. Therefore, this refactoring is a behavior preserving transformation.

Precondition (a) is obvious. The actor A_1 represents the general role of the actor A. This is expressed by goals $\{G_1, G_2, \cdots, G_n\}$. Precondition (b) ensures unique parent (invariant one in Section 3.3). Precondition (c) ensures distinct environmental entity name (invariant two in Section 3.3).

Postcondition (a) ensures that the new actor A_1 is created properly. It is an abstract actor. The actor A_1 has goals $\{G_1, G_2, \cdots, G_n\}$. The actor A inherits these goals from A_1. Thus the actor A_1 can be reused by other actors during the evolution of the use case model.

A.6 Move an Environmental Entity

A.6.1 Move_Goal_To_Parent_Actor

Description: This refactoring moves a goal from child actor(s) to its parent actor. It helps improve the understandability and reusability of the actor.

Arguments: actor A, the set of its child actors $\{a_1, a_2, \cdots, a_n\}$, goal g to be moved from child actors to actor A.

Preconditions:

(a). The actor A is an abstract actor. It does not have the goal g. The actor A is the parent of actors $\{a_1, a_2, \cdots, a_n\}$.

$$\begin{aligned}
&A \in Model.actors \wedge A.type = abstract \\
&\wedge (\forall a \in \{a_1, a_2, \cdots, a_n\}, a \in Model.actors \wedge a.parentActor = A)
\end{aligned}$$

(b). The actor A is not referenced by any other actor than $\{a_1, a_2, \cdots, a_n\}$.

$$A.actors = \{a_1, a_2, \cdots, a_n\}$$

(c). Every actor in $\{a_1, a_2, \cdots, a_n\}$ has the goal g.

$$\forall a \in \{a_1, a_2, \cdots, a_n\}, g \in a.goals.$$

Postconditions:

(a). The goal g is removed from actors $\{a_1, a_2, \cdots, a_n\}$ and moved to the actor A.

$$\begin{aligned}&(\forall a \in \{a_1, a_2, \cdots, a_n\}, g \notin a.goals)\\&\land\, g \in A.goals\end{aligned}$$

Verification: This refactoring moves a goal from child actors $\{a_1, a_2, \cdots, a_n\}$ to the parent actor A. After the refactoring, child actors inherit the goal g from the actor A. The relationship between the goal model and these child actors is preserved.

Preconditions (a) and (c) are obvious. Precondition (b) ensures that the actor A and its child actors are isolated from other actors. Otherwise, moving the goal g may affect other actors.

B. Refactorings at the Structure Level

B.1 Create a Structural Entity

B.1.1 Create_Empty_Episode

Description: create a new episode without any defined event. The new episode is isolated from other episodes.

Arguments: string newEpisodeName

Preconditions:

(a). The name of the new episode does not clash with an already existing episode within the use case model.

$$\forall episode \in Model.episodes, episode.name \neq newEpisodeName.$$

Postconditions:

(a). The new episode is empty and is isolated from other episodes.

$$\exists e \in Model'.episodes \wedge e \notin Model.episodes \wedge e.name = newEpisodeName$$
$$\wedge\ e.episodes = \{\} \wedge e.events = \{\}$$

Verification: Since the new episode is isolated from other episodes, it does not participate in the interaction between use cases and actors. Therefore, this refactoring does not change the behavior of the use case model. Precondition (a) ensures distinct episode name (invariant three in Section 3.3).

B.2 Delete a Structural Entity

B.2.1 Delete_Unreferenced_Episode

Description: delete an unreferenced episode from the use case model.

Arguments: episode E.

Preconditions:

(a). The episode E is not referenced by any use case or any other episode.

$$E \in Model.episodes \wedge E.usecases = \{\} \wedge E.episodes = \{\}$$

Postconditions:

(a). The episode E is deleted from the use case model.

$$E \notin Model'.episodes$$

(b). The deletion is cascaded to the event level. After the episode E is deleted from the use case model, some events may be unreferenced. These events are deleted.

$$\forall v \in E.events \wedge v.episodes = \{E\}, v \notin Model'.events.$$

Verification: Since the episode E is isolated from use cases and other episodes, it does not participate in the interaction between use cases and actors. Deleting it does not change the behavior of the use case model.

B.3 Change a Structural Entity

B.3.1 Change_Episode_Name

Description: change the name of an episode in the use case model.

Arguments: episode E, newEpisodeName

Preconditions:

(a). $E \in Model.episodes$

(b). The new episode name does not clash with an already existing episode within the use case model.

$$\forall episode \in Model.episodes, episode.name \neq newEpisodeName.$$

Postconditions:

(a). Episode E is given a new name (newEpisodeName).

$$E.name = newEpisodeName$$

Verification: Precondition (a) is obvious. Precondition (b) ensures distinct episode name (invariant three in Section 3.3). Changing the name of an episode does not change the behavior of the use case model.

B.4 Compose a Structural Entity

B.4.1 Episode_Sequence_Merger

Description: When there is a *sequence* relationship between two primitive episodes, this refactoring can be used to merge them into a new primitive episode. It helps manage the episode granularity and improve the use case internal structure.

Arguments: primitive episode E_1, primitive episode E_2, string newEpisodeName.

Preconditions:

(a). There is a *sequence* relationship between episodes E_1 and E_2. E_2 follows E_1 immediately.

$$\begin{aligned}&E_1 \in Model.episodes \land E_2 \in Model.episodes\\&\land E_1.type = PrimitiveEpisode \land E_2.type = PrimitiveEpisode\\&\land Sequence\ [E_1, E_2]\end{aligned}$$

(b). The name of the new episode does not clash with an already existing episode within the use case model.

$$\forall episode \in Model.episodes, episode.name \neq newEpisodeName.$$

Postconditions:

(a). The new primitive episode is created by merging episodes E_1 and E_2 together.

$$\begin{aligned}&\exists E \notin Model.episodes \land E \in Model'.episodes\\&\land E.name = newEpisodeName \land E.type = PrimitiveEpisode\\&\land T[E] = T[E_1 \cdot E_2]\end{aligned}$$

(b). All occurrences with the *sequence* relationship between E_1 and E_2 in the use case model are replaced by the new episode E.

$$\forall u \in Model.usecases, SubstituteEpisodeTreeInUsecase(u, T[E_1 \cdot E_2], T[E]).$$

(c). After the above step, if episodes E_1 and E_2 are not referenced by any use case, they are deleted from the use case model.

$$(E_1.usecases = \{\} \Rightarrow E_1 \notin Model'.episodes) \wedge$$
$$(E_2.usecases = \{\} \Rightarrow E_2 \notin Model'.episodes)$$

Verification: According to the definition of the *sequence* relationship between episodes in Section 3.1.3, it is apparent that this refactoring is a behavior preserving transformation. The process algebra term of the new primitive episode E equals to the one of episodes E_1 and E_2: $P_E = P_{E_1} \cdot P_{E_2}$.

Precondition (a) specifies the *sequence* relationship between episodes E_1 and E_2. Precondition (b) ensures distinct episode name (invariant three in Section 3.3).

Postcondition (a) specifies the episode model of the new episode. The new episode has exactly the same behavior as the combination of episode E_1 and E_2. Postcondition (b) ensures that this refactoring is applied to all occurrences consistently where there is a *sequence* relationship between episodes E_1 and E_1. If only one of episodes E_1 and E_2 is used by a certain use case, this refactoring is not applicable in this particular case. After the refactoring, unreferenced episodes should be removed. This is specified in Postcondition (c).

B.4.2 Episode_Alternation_Merger

Description: When there is an *alternation* relationship between two primitive episodes, this refactoring can be used to merge them into a new primitive episode. It helps manage the episode granularity and improve the use case internal structure.

Arguments: episode E_1, episode E_2, string newEpisodeName.

Preconditions:

(a). There is an *alternation* relationship between episodes E_1 and E_2.

$$\begin{aligned}&E_1 \in Model.episodes \wedge E_2 \in Model.episodes\\ &\wedge E_1.type = PrimitiveEpisode \wedge E_2.type = PrimitiveEpisode\\ &\wedge Alternation\ [E_1, E_2]\end{aligned}$$

(b). The name of the new episode does not clash with an already existing episode within the use case model.

$$\forall episode \in Model.episodes, episode.name \neq newEpisodeName.$$

Postconditions:

(a). The new primitive episode E is created by merging episodes E_1 and E_2 together.

$$\begin{aligned}&\exists E \notin Model.episodes \wedge E \in Model'.episodes\\&\wedge E.name = newEpisodeName \wedge E.type = PrimitiveEpisode\\&\wedge T[E] = T[E_1 + E_2]\end{aligned}$$

(b). All occurrences with the *alternation* relationship between episodes E_1 and E_2 in the use case model are replaced by the new episode E.

$$\forall u \in Model.usecases, SubstituteEpisodeTreeInUsecase(u, T[E_1 + E_2], T[E]).$$

(c). After the above step, if episodes E_1 and E_2 are not referenced by any use case, they are deleted from the use case model.

$$\begin{aligned}&(E_1.usecases = \{\} \Rightarrow E_1 \notin Model'.episodes) \wedge\\&(E_2.usecases = \{\} \Rightarrow E_2 \notin Model'.episodes)\end{aligned}$$

Verification: According to the definition of the *alternation* relationship between episodes in Section 3.1.3, it is apparent that this refactoring is a behavior preserving transformation. The process algebra term of the new primitive episode E equals to the one of episodes E_1 and E_2: $P_E = P_{E_1} + P_{E_2}$.

Precondition (a) specifies the relationship between episodes E_1 and E_2. Precondition (b) ensures ensures distinct episode name (invariant three in Section 3.3).

Postcondition (a) specifies the episode model of the new episode. The new episode has exactly the same behavior as the combination of episode E_1 and E_2. Postcondition (b) ensures that this refactoring applies to all occurrences consistently where there is an *alternation* relationship between episodes E_1 and E_1. If only one of E_1 and E_2 is used by a certain use case, this refactoring is not applicable in this particular case. After the refactoring, unreferenced episodes should be removed. This is specified in Postcondition (c).

B.4.3 Episode_Parallel_Merger

Description: When there is a *parallel* relationship between two episodes, this refactoring can be used to merge them into a new primitive episode. It helps manage the episode granularity and improve the use case internal structure.

Arguments: episode E_1, episode E_2, string newEpisodeName.

Preconditions:

(a). There is a *parallel* relationship between episodes E_1 and E_2.

$$\begin{aligned}&E_1 \in Model.episodes \wedge E_2 \in Model.episodes\\ &\wedge E_1.type = PrimitiveEpisode \wedge E_2.type = PrimitiveEpisode\\ &\wedge Parallel\,[E_1, E_2]\end{aligned}$$

(b). The name of the new episode does not clash with an already existing episode within the use case model.

$$\forall episode \in Model.episodes, episode.name \neq newEpisodeName.$$

Postconditions:

(a). The new primitive episode E is created by merging episodes E_1 and E_2 together.

$$\begin{aligned}&\exists E \notin Model.episodes \wedge E \in Model'.episodes\\ &\wedge E.name = newEpisodeName \wedge E.type = PrimitiveEpisode\\ &\wedge T[E] = T[E_1 \parallel E_2]\end{aligned}$$

(b). All occurrences with the *parallel* relationship between episodes E_1 and E_2 in the use case model are replaced by the new episode E.

$$\forall u \in Model.usecases, SubstituteEpisodeTreeInUsecase(u, T[E_1 \parallel E_2], T[E]).$$

(c). After the above step, if episode E_1 and E_2 are not referenced by any use case, they are deleted from the use case model.

$$\begin{aligned}&(E_1.usecases = \{\} \Rightarrow E_1 \notin Model'.episodes)\wedge\\ &(E_2.usecases = \{\} \Rightarrow E_2 \notin Model'.episodes)\end{aligned}$$

Verification: According to the definition of the *parallel* relationship between episodes in Section 3.1.3, it is apparent that this refactoring is a behavior preserving transformation. The process algebra term of the new primitive episode E equals to the one of episode E_1 and E_2: $P_E = P_{E_1} \parallel P_{E_2}$.

Precondition (a) specifies the relationship between episodes E_1 and E_2. Precondition (b) ensures ensures distinct episode name (invariant three in Section 3.3).

Postcondition (a) specifies the episode model of the new episode. The new episode has exactly the same behavior as the combination of episode E_1 and E_2. Postcondition (b) ensures that this refactoring applies to all occurrences consistently where there is a *parallel* relationship between episodes E_1 and E_1. If only one of E_1 and E_2 is used by a certain use case, this refactoring is not applicable in this particular case. After the refactoring, unreferenced episodes should be removed. This is specified in Postcondition (c).

B.4.4 Generalize_Episodes

Description: The Generalize_Episodes refactoring is suggested when two or more episodes have a common event subtree. After this refactoring, an abstract episode is created as the parent of these episodes. The common event subtree is moved into the new parent episode. This refactoring helps reduce the redundancy and improve the reusability.

Arguments: a set of episodes $\{E_1, E_2, \cdots, E_n\}$, a common event subtree t, string newEpisodeName.

Preconditions:

(a). Episodes $\{E_1, E_2, \cdots, E_n\}$ have a common event subtree t.

$$\forall E \in \{E_1, E_2, \cdots, E_n\}, E \in Model.episodes \land IsEventSubTree(E.eventTree, t).$$

(b). Episodes $\{E_1, E_2, \cdots, E_n\}$ do not have any existing parent episode.

$$\forall E \in \{E_1, E_2, \cdots, E_n\}, \nexists E.parentEpisode.$$

(c). The name of the new episode does not clash with an already existing episode within the use case model.

$$\forall episode \in Model.episodes, episode.name \neq newEpisodeName.$$

Postconditions:

(a). A new abstract episode E is created with the name of newEpisodeName. It becomes the parent of episodes $\{E_1, E_2, \cdots, E_n\}$. The event tree of E is the subtree t.

$$\exists E \notin Model.episodes \wedge E \in Model'.episodes \wedge E.name = newEpisodeName$$
$$\wedge\ E.type = abstract \wedge E.eventTree = t$$
$$\wedge\ (\forall e \in \{E_1, E_2, \cdots, E_n\}, e.parentEpisode = E)$$

(b). In episodes $\{E_1, E_2, \cdots, E_n\}$, events occurring in the event subtree t are replaced by corresponding event placeholders.

$$\forall v \in t.events \wedge \forall e \in \{E_1, E_2, \cdots, E_n\}, SubstituteEventWithPlaceholder(e, v).$$

Verification: This refactoring reuses the common behavior by moving up the common event sequence to the parent episode. Since the same event sequence is inherited in child episodes, the behavior of the use case model is preserved.

Precondition (a) ensures that episodes $\{E_1, E_2, \cdots, E_n\}$ have a common event subtree. Precondition (b) avoids multiple inheritances. This ensures unique parent (invariant one in Section 3.3). Precondition (c) ensures distinct episode name (invariant three in Section 3.3).

Postcondition (a) ensures that an abstract episode is created. This episode does not participate in the episode tree directly. Instead, its child episodes inherit its behavior and form the episode tree. Postcondition (b) ensures that related events are replaced by corresponding event placeholders properly in child episodes.

B.5 Decompose a Structural Entity

B.5.1 Episode_Sequence_Generation

Description: This refactoring splits one primitive episode into two primitive episodes and generates a *sequence* relationship between these two episodes. When the event model of an episode is too complicated, it is preferable to split the event tree. After this refactoring, a new *sequence* composite episode is created to replace the original episode. This refactoring helps reduce the episode granularity and improve the understandability.

Arguments: primitive episode E, its event subtrees T_1 and T_2, string newEpisodeName1, string newEpisodeName2.

Preconditions:

(a). $E \in Model.episodes \wedge E.type = PrimitiveEpisode \wedge$
$newEpisodeName1 \neq newEpisodeName2$

(b). The event tree of E can be represented by T_1 and T_2 with the *sequence* relationship.

$$\exists x \wedge \exists y \wedge E.eventTree = T[x \cdot y] \wedge T[x] = T_1 \wedge T[y] = T_2$$

(c). The name of new episodes does not clash with an already existing episode within the use case model.

$$\begin{aligned} \forall episode \in Model.episodes, episode.name \neq newEpisodeName1 \wedge \\ episode.name \neq newEpisodeName2. \end{aligned}$$

Postconditions:

(a). New episodes E_1 and E_2 are created with the event subtrees T_1 and T_2 respectively.

$$\begin{aligned} &E_1 \notin Model.episodes \wedge E_1 \in Model'.episodes \wedge E_1.name = newEpisodeName1 \wedge \\ &E_2 \notin Model'.episodes \wedge E_2 \in Model'.episodes \wedge E_2.name = newEpisodeName2 \wedge \\ &E_1.type = PrimitiveEpisode \wedge E_2.type = PrimitiveEpisode \wedge \\ &E_1.eventTree = T_1 \wedge E_2.eventTree = T_2 \end{aligned}$$

(b). The episode E is replaced by a sequence composite episode, which contains episodes E_1 and E_2. E_2 follows E_1 immediately.

$$\begin{aligned} &\exists E' \in Model'.episodes \wedge E'.type = SequenceEpisode \\ &\wedge T[E'] = T[E_1 \cdot E_1] \\ &\wedge SubstituteEpisode(E, E') \end{aligned}$$

Verification: Conversely to the *Episode_Sequence_Merger* refactoring, this refactoring splits one primitive episode into two primitive episodes and generates a *sequence* relationship between these two episodes. According to the definition of the *sequence* relationship between episodes in Section 3.1.3, it is apparent that this refactoring is a behavior preserving transformation. After this refactoring, both new primitive episodes can be merged back so that the original primitive episode is restored.

Precondition (a) is obvious. Precondition (b) specifies the pattern of the event tree of the original episode E. Precondition (c) ensures distinct episode name (invariant three in Section 3.3).

Postcondition (a) ensures that new primitive episodes are created with event subtrees specified in arguments. So new primitive episodes have the equivalent behavior as the original primitive episode. Postcondition (b) ensures that the original primitive episode E is replaced by a *sequence* composite episode E', which is composed of new primitive episodes. All occurrences of the episode E must be replaced. Therefore, the episode E is restructured and the change is applied to the use case model consistently.

B.5.2 Episode_Alternation_Generation

Description: This refactoring splits one primitive episode into two primitive episodes and generates an *alternation* relationship between these two primitive episodes. When the event model of an episode is too complicated, it is preferable to split the event tree. After this refactoring, a new *alternation* composite episode is created to replace the original episode. This refactoring helps reduce the episode granularity and improve the understandability.

Arguments: primitive episode E, its event subtrees T_1 and T_2, string newEpisodeName1, string newEpisodeName2.

Preconditions:

(a). $E \in Model.episodes \wedge E.type = PrimitiveEpisode \wedge$
$newEpisodeName1 \neq newEpisodeName2$

(b). The event tree of E can be represented by T_1 and T_2 with the *alternation* relationship.

$$\exists x \wedge \exists y \wedge E.eventTree = T[x + y] \wedge T[x] = T_1 \wedge T[y] = T_2$$

(c). The name of new episodes does not clash with an already existing episode within the use case model.

$$\forall episode \in Model.episodes, episode.name \neq newEpisodeName1 \wedge$$
$$episode.name \neq newEpisodeName2.$$

Postconditions:

(a). Episodes E_1 and E_2 are created with event subtrees T_1 and T_2 respectively.

$$\begin{aligned}
&E_1 \notin Model.episodes \wedge E_1 \in Model'.episodes \wedge E_1.name = newEpisodeName1 \wedge \\
&E_2 \notin Model'.episodes \wedge E_2 \in Model'.episodes \wedge E_2.name = newEpisodeName2 \wedge \\
&E_1.type = PrimitiveEpisode \wedge E_2.type = PrimitiveEpisode \wedge \\
&E_1.eventTree = T_1 \wedge E_2.eventTree = T_2
\end{aligned}$$

(b). The episode E is replaced by an alternation composite episode, which contains episodes E_1 and E_2.

$$\begin{aligned}
&\exists E' \in Model'.episodes \wedge E'.type = AlternationEpisode \\
&\wedge T[E'] = T[E_1 + E_1] \\
&\wedge SubstituteEpisode(E, E')
\end{aligned}$$

Verification: Conversely to the *Episode_Alternation_Merger* refactoring, this refactoring splits one primitive episode into two primitive episodes and generates an *alternation* relationship between these primitive episodes. According to the definition of the *alternation* relationship between episodes in Section 3.1.3, it is apparent that this refactoring is a behavior preserving transformation. After this refactoring, both new primitive episodes can be merged back so that the original primitive episode is restored.

Precondition (a) is obvious. Precondition (b) specifies the pattern of the event tree of the original episode E. Precondition (c) ensures distinct episode name (invariant three in Section 3.3).

Postcondition (a) ensures that new primitive episodes are created with event subtrees specified in arguments. So new primitive episodes have the equivalent behavior as the original primitive episode. Postcondition (b) ensures that the original primitive episode E is replaced by an *alternation* composite episode E', which is composed of new primitive episodes. All occurrences of the episode E must be replaced. Therefore, the episode E is restructured and the change is applied to the use case model consistently.

B.5.3 Episode_Parallel_Generation

Description: This refactoring splits one primitive episode into two primitive episodes and generates a *parallel* relationship between these two episodes. When the event model of an episode is too complicated, it is preferable to split the event tree. After this refactoring, a new *parallel* composite episode is created to replace the original episode. This refactoring helps reduce the episode granularity and improve the understandability.

Arguments: primitive episode E, its event subtrees T_1 and T_2, string newEpisodeName1, string newEpisodeName2.

Preconditions:

(a). $E \in Model.episodes \wedge E.type = PrimitiveEpisode \wedge$
$newEpisodeName1 \neq newEpisodeName2$

(b). The event tree of E can be represented by T_1 and T_2 with the *parallel* relationship.

$$\exists x \wedge \exists y \wedge E.eventTree = T[x \parallel y] \wedge T[x] = T_1 \wedge T[y] = T_2$$

(c). The name of new episodes does not clash with an already existing episode within the use case model.

$$\forall episode \in Model.episodes, episode.name \neq newEpisodeName1 \wedge \\ episode.name \neq newEpisodeName2.$$

Postconditions:

(a). Episodes E_1 and E_2 are created with event subtrees T_1 and T_2 respectively.

$$E_1 \notin Model.episodes \wedge E_1 \in Model'.episodes \wedge E_1.name = newEpisodeName1 \wedge \\ E_2 \notin Model'.episodes \wedge E_2 \in Model'.episodes \wedge E_2.name = newEpisodeName2 \wedge \\ E_1.type = PrimitiveEpisode \wedge E_2.type = PrimitiveEpisode \wedge \\ E_1.eventTree = T_1 \wedge E_2.eventTree = T_2$$

(b). The episode E is replaced by a parallel composite episode, which contains episodes E_1 and E_2.

$$\exists E' \in Model'.episodes \wedge E'.type = ParallelEpisode \\ \wedge T[E'] = T[E_1 \parallel E_1] \\ \wedge SubstituteEpisode(E, E')$$

Verification: Conversely to the *Episode_Parallel_Merger* refactoring, this refactoring splits one primitive episode into two primitive episodes and generates a *parallel* relationship between these two primitive episodes. According to the definition of the *parallel* relationship between episodes in Section 3.1.3, it is apparent that this refactoring is a behavior preserving transformation. After this refactoring, both new primitive episodes can be merged back so that the original primitive episode is restored.

Precondition (a) is obvious. Precondition (b) specifies the pattern of the event tree of the original episode E. Precondition (c) ensures distinct episode name (invariant three in Section 3.3).

Postcondition (a) ensures that new primitive episodes are created with event subtrees specified in arguments. So new primitive episodes have the equivalent behavior as the original primitive episode. Postcondition (b) ensures that the original primitive episode E is replaced by a *parallel* composite episode E', which is composed of new primitive episodes. All occurrences of the episode E must be replaced. Therefore, the episode E is restructured and the change is applied to the use case model consistently.

B.6 Move a Structural Entity

B.6.1 Move_Episode_To_Parent_Usecase

Description: This refactoring moves an episode from child use cases to the parent use case. It helps improve the use case reusability and understandability.

Arguments: use case U, its child use cases $\{u_1, u_2, \cdots, u_n\}$, a common episode e to be moved.

Preconditions:

(a). $U \in Model.usecases \wedge (\forall u \in \{u_1, u_2, \cdots, u_n\}, u \in Model.usecases)$

(b). The use case U is an abstract use case in the use case model.

$$U.type = abstract$$

(c). The use case U is the parent of use cases $\{u_1, u_2, \cdots, u_n\}$. It does not have any other child use case.

$$(\nexists u \in \{u_1, u_2, \cdots, u_n\} \wedge u.parentUsecase \neq U) \wedge$$
$$(\nexists u \notin \{u_1, u_2, \cdots, u_n\} \wedge u.parentUsecase = U)$$

(d). The episode e is a common episode within use cases $\{u_1, u_2, \cdots, u_n\}$.

$$\nexists u \in \{u_1, u_2, \cdots, u_n\} \wedge e \notin u.episodes$$

Postconditions:

(a). The episode e is moved to the use case U. In use cases $\{u_1, u_2, \cdots, u_n\}$, the episode e is replaced by an episode placeholder.

$$e \in U' \wedge (\forall u \in \{u'_1, u'_2, \cdots, u'_n\}, SubstituteEpisodeWithPlaceholder(u, e)).$$

Verification: This refactoring moves a common episode from child use cases to the parent use case. Since this common episode is inherited in child use cases after the refactoring, the behavior of the use case model is preserved.

B.7 Convert a Structural Entity

B.7.1 Convert_Episode_To_Usecase

Description: When an episode is too complicated, some refactorings can be used to split it into two episodes. The *Convert_Episode_to_Usecase* refactoring, however, reduces the complexity using another approach. It converts the entire episode into a new use case. This refactoring helps reduce the use case granularity and improve the understandability.

Arguments: original use case U, episode e to be converted, string newUsecaseName.

Preconditions:

(a). The use case U contains the episode e. The episode e is not referenced by any other use case.

$$U \in Model.usecases \wedge e \in U.episodes \wedge e.usecases = \{U\}$$

(b). The name of the new use case does not clash with an already existing use case within the use case model.

$$\forall usecase \in Model.usecases, usecase.name \neq newUsecaseName.$$

(c). One of the following conditions is satisfied.

(c1). The episode e is in the normal execution path and it is executed at the beginning or at the end.

$$\exists x \wedge p_x \neq \varepsilon \wedge (U.episodeTree = T[e \cdot x] \vee U.episodeTree = T[x \cdot e])$$

(c2). The episode e is in the normal execution path and it is not executed at the beginning or at the end, the use case U includes the new use case.

$$\exists x \wedge \exists y \wedge p_x \neq \varepsilon \wedge p_y \neq \varepsilon \wedge U.episodeTree = T[x \cdot e \cdot y]$$

(c3). The parent node of the episode is an *alternation* composite episode located at the root node of the episode tree.

$$\exists x \wedge (U.episodeTree = T[e + x])$$

Postconditions:

(a). A new use case U_1 is created with the name of newUsecaseName. The episode e is moved into this use case.

$$\begin{aligned}&\exists U_1 \notin Model.usecases \wedge U_1 \in Model'.usecases\\ &\wedge U_1.type = abstract \wedge U_1.name = newUsecaseName \wedge e \in U_1.episodes\end{aligned}$$

(b). Depending on which condition defined in Precondition (c) is met, a different relationship is generated between the use case U and the new use case U_1. Different situations are analysed as follows in the order of conditions defined in Precondition (c).

(c1). If the episode e is in the normal execution path and it is executed at the beginning or at the end, a precedence relationship is created.

$$\begin{aligned}U.episodeTree = T[e \cdot x] &\Rightarrow Precedence[U_1, U] \wedge U_1.actors = U.actors\\ &\wedge U_1.type = abstract.\\ U.episodeTree = T[x \cdot e] &\Rightarrow Precedence[U, U_1] \wedge U_1.actors = U.actors\\ &\wedge U_1.type = abstract.\end{aligned}$$

(c2). If the episode e is in the normal execution path and it is not executed at the beginning or at the end, the use case U includes the new use case U_1.

$$U_1.type = abstract \wedge Inclusion\ [U, U_1]$$
$$\wedge\ SubstituteEpisodeWithPseudo(U, e, Pseudo\ [U_1])$$

(c3). If the parent node of the episode e is an *alternation* composite episode located at the root node of the episode tree, the new use case U_1 is independent to the use case U'.

$$U_1 \notin U'.usecases \wedge e \notin U'.episodes \wedge U_1.actors = U'.actors$$

Verification: The semantics of the use case relationship is analysed in Section 3.2.1. After this refactoring, these two use cases can be merged back so that the original use case is restored. Therefore, it is a behavior preserving transformation.

Precondition (a) isolates the use case U from other use cases. This is necessary because some behavior is moved to the new use case. The behavior of other use cases may be changed if they have references to the original use case U. Precondition (b) ensures distinct episode name (invariant three in Section 3.3).

Postcondition (a) specifies how the new use case is created. Postcondition (b) specifies how the relationship between the use case U' and the new use case U_1 should be generated. If the relationship is not generated properly, merging two use cases back will not be able to restore the original state. The behavior of the use case model may be changed.

C. Refactorings at the Event Level

C.1 Create an Event

C.1.1 Create_Unreferenced_Event

Description: create a new event without participating in any event model.

Arguments: string newEventName.

Preconditions:

(a). The name of the new event does not clash with an already existing event within the use case model.

$$\forall event \in Model.events, event.name \neq newEventName.$$

Postconditions:

(a). The new event is created. It is isolated from other events in the use case model.

$$\exists e \in Model'.events \wedge e \notin Model.events \wedge e.name = newEventName \wedge e.events = \{\}$$

Verification: Precondition (a) ensures distinct event name (invariant four in Section 3.3). Since the new event is isolated from other events, it does not affect the behavior of the use case model. This refactoring is a behavior preserving transformation.

C.2 Delete an Event

C.2.1 Delete_Unreferenced_Event

Description: delete an unreferenced event from the use case model.

Arguments: event v.

Preconditions:

(a). The event v is not referenced by any episode.

$$v \in Model.events \wedge v.episodes = \{\}$$

Postconditions:

(a). The event v is deleted from the use case model.

$$v \notin Model'.events$$

Verification: Since the event v does not participate in the event model of any episode, it does not affect the interaction between actors and use cases. In other words, it does not affect the behavior of the use case model. Deleting it is a behavior preserving transformation.

C.3 Change an Event

C.3.1 Change_Event_Name

Description: change the name of an event within the use case model.

Arguments: event v, string newEventName

Preconditions:

(a). $v \in Model.events$

(b). The new event name does not clash with an already existing event within the use case model.

$$\forall event \in Model.events, event.name \neq newEventName.$$

Postconditions:

(a). The event v is given a new name (newEventName).

$$e.name = newEventName$$

Verification: Precondition (a) ensures distinct event name (invariant four in Section 3.3). Changing the name of an event does not change the behavior of the use case model.