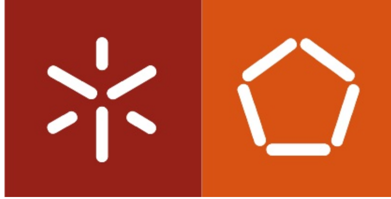
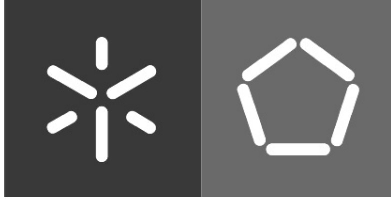


Universidade do Minho
Escola de Engenharia

Rui Filipe Pereira de Azevedo

Systematic Method for UML Model to Model Transformation: Development and Verification in Alloy

Abril de 2012



Universidade do Minho
Escola de Engenharia

Rui Filipe Pereira de Azevedo

Systematic Method for UML Model to Model Transformation: Development and Verification in Alloy

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação de
Professor Doutor Fernando Mário Junqueira Martins

Abril de 2012

Declaração

Nome: Rui Filipe Pereira de Azevedo

Endereço Electrónico: rfpazevedo@gmail.com

Telefone: 964965426

Bilhete de Identidade: 13009435

Título da Tese: Systematic Method for UML Model to Model Transformation: Development and Verification in Alloy

Orientador: Professor Doutor Fernando Mário Junqueira Martins

Ano de conclusão: 2012

Designação do Mestrado: Mestrado em Engenharia Informática

É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE.

Universidade do Minho, 27 de Abril de 2012

Rui Filipe Pereira de Azevedo

Abstract

The Unified Modeling Language (UML) is nowadays the industry standard notation for modelling software systems using an object-oriented approach. The Object Management Group (OMG) manages this standardization. UML combines several modelling techniques and its models have visual representations through UML diagrams. Despite being widely accepted, used and also recommended by software development processes like Rational Unified Process (RUP) and Agile, two major UML weaknesses are recognized by the overall software community: it is a notation with no underlying method; it is only semi-formal.

Trying to narrow this gap, this work presents a method for the systematic transformation of UML models. Furthermore, and tackling another vulnerability of UML, its informality, we also propose a verification mechanism for checking the correctness of said transformations using the Alloy formal modelling notation.

The proposed diagram transformation method follows the RUP use case orientation and encompasses three UML diagrams: use case, sequence, and interaction overview diagrams.

We have developed the *action step* and *action block* constructs, which will be the basis for a more precise and standardized structure for the textual specification of use cases. Using these constructs, and without loss of expressive power, a canonical form for use cases was devised which will be the source and the anchor for the other steps of the systematic transformation method. Starting from the use cases already in the canonical form, we have created a set of steps and rules that will conduct the transformation of these use cases into sequence and interaction overview diagrams in a systematic way. With Alloy, we are able to assess the diagrams' well-formedness and verify the correction of the transformations.

Resumo

A Unified Modeling Language (UML) é hoje em dia a notação standard da indústria para a modelação de sistemas de *software* usando uma abordagem orientada aos objectos. O Object Management Group (OMG) gere esta standardização. A UML combina várias técnicas de modelação e os seus modelos têm representações visuais através de diagramas. Apesar de ser amplamente aceite, usada e também recomendada por processos de *software* como o Rational Unified Process (RUP) e *Agile*, duas fragilidades são reconhecidas à UML pela comunidade de *software* em geral: é uma notação sem método subjacente; é apenas semi-formal.

Tentando estreitar esta lacuna, este trabalho apresenta um método para a transformação sistemática de modelos UML. Para além disso, e abordando outra vulnerabilidade da UML, a informalidade, propomos também um mecanismo de verificação da correcção das referidas transformações usando a notação de modelação formal Alloy.

O método de transformação de diagramas proposto segue a orientação aos casos de uso do RUP e abarca três diagramas UML: diagramas de casos de uso, de sequência, e de supervisão de interação.

Desenvolvemos as construções *passos de ação* e *blocos de ações*, as quais serão a base para uma estrutura mais precisa e standardizada das especificações dos casos de uso. Usando estas construções, e sem perda de poder expressivo, foi concebida uma forma canónica para os casos de uso que será a origem e a âncora para os outros passos do método sistemático de transfomação. Partindo dos casos de uso já na forma canónica, criamos um conjunto de passos e regras que conduzirão a transformação destes casos de uso para diagramas de sequência e de supervisão de interação de um modo sistemático. Através do Alloy, somos capazes de aferir a boa-formação dos diagramas e verificar a correção das transformações.

Contents

List of Tables ix

List of Figures xiii

List of Acronyms xvi

1 Introduction 1

1.1 Motivation . . . 2

1.2 Contributions . . . 3

1.3 Outline . . . 5

2 Related Work 6

2.1 Unified Modeling Language . . . 6

2.1.1 Use Cases . . . 7

2.1.2 System Sequence Diagrams . . . 15

2.1.3 Interaction Overview Diagrams . . . 17

2.2 Transformation and Consistency of UML Models . . . 18

2.2.1 General Terms . . . 19

2.2.2 Consistency Approaches . . . 20

2.3 Formal Methods . . . 22

2.3.1 OCL . . . 23

2.3.2 SPIN model checker 24
2.3.3 Z 24
2.3.4 Graph Transformation 25
2.3.5 Alloy 26
2.4 Conclusion 28

3 Alloy 29

3.1 Atoms and Relations 31
3.2 Object-Oriented and Set Theoretic Views 33
3.3 Language 33
3.3.1 Signatures 34
3.3.2 Constraints 37
3.3.3 Commands and Scope 42
3.4 Alloy Analyzer 43
3.5 Conclusion 45

4 Alloy Models of UML Diagrams 46

4.1 Canonical Form Use Cases 48
4.1.1 Towards a Canonical Form for Use Cases 49
4.1.2 Use Case Diagram 59
4.1.3 Textual Specification of Use Cases 68
4.2 System Sequence Diagrams 77
4.2.1 Actor and System 79
4.2.2 Message 79
4.2.3 Ref 80
4.2.4 Frames 80
4.2.5 System Sequence Diagram 82
4.3 Interaction Overview Diagram 83

4.3.1 Interaction Overview Diagram 83
4.3.2 Initial Node 84
4.3.3 Decision Node 85
4.3.4 Ref 87
4.3.5 Sequence Diagram 87
4.3.6 Activity and Flow Final Nodes 87
4.4 Conclusion 88

5 Transformation Rules 89
5.1 From Use Cases to System Sequence Diagrams 89
5.1.1 Action Steps 90
5.1.2 Alternative Flows and Exceptions 95
5.1.3 Include and Extends Relations 99
5.2 From Use Cases to Interaction Overview Diagrams 100
5.2.1 Action Blocks 101
5.2.2 Alternative Flows and Exceptions 104
5.2.3 Include and Extends Relations 106
5.3 Application of the Transformation Rules 107
5.4 Conclusion 114

6 Case Study 116
6.1 Domain 116
6.2 From Use Cases to System Sequence Diagrams 119
6.2.1 Step One 119
6.2.2 Step Two 122
6.2.3 Step Three 122
6.2.4 Step Four 123
6.2.5 Step Five 124

6.2.6 Step Six . . . 126
6.2.7 Step Seven . . . 126
6.3 From Use Cases to Interaction Overview Diagrams . . . 128
6.3.1 Step One . . . 128
6.3.2 Step Two . . . 129
6.3.3 Step Three . . . 130
6.3.4 Step Four . . . 131
6.3.5 Step Five . . . 131
6.3.6 Step Six . . . 132
6.4 Consistency Verification . . . 133
6.5 Conclusion . . . 135

7 Conclusion and Future Work 137

Bibliography 140

A Case Study Diagrams 148
A.1 Textual Use Cases . . . 148
A.2 System Sequence Diagrams . . . 151
A.3 Interaction Overview Diagrams . . . 153

List of Tables

4.1 Textual representation of the Order Product use case with Assume action steps. . . . 52

4.2 Structure of the different kinds of action blocks. . . . 56

4.3 New textual representation of the Order Product use case without Assume action steps and in double-column format. . . . 58

4.4 Entity *Actor*. . . . 59

4.5 Well-formedness rule #1: Generalization relation between actors is acyclic. . . . 60

4.6 Entity *UseCase*. . . . 61

4.7 Well-formedness rule #2: A use case cannot include nor extend its parent use case. . . . 61

4.8 Well-formedness rule #3: Specializing use cases are extended by all use cases that extend their parent. . . . 62

4.9 Well-formedness rule #4: Abstract use cases are specialized by at least two use cases. . . . 63

4.10 Well-formedness rule #5: Generalization relation between use cases is acyclic. . . . 64

4.11 Well-formedness rule #6: Inclusion relation is acyclic. . . . 64

4.12 Well-formedness rule #7: Actors cannot be associated with included use cases. . . . 66
4.13 Entity *UseCaseModel*. . . . 66
4.14 Well-formedness rule #8: A use case cannot be associated with two actors related with inheritance. . . . 67
4.15 Well-formedness rule #9: An actor cannot be associated with two use cases related with inheritance. . . . 67
4.16 Entity *Step*. . . . 68
4.17 Entity *Atom*. . . . 68
4.18 Entity *Goto*. . . . 69
4.19 Entity *Include*. . . . 69
4.20 Entity *Specialize*. . . . 70
4.21 Entity *Choice*. . . . 71
4.22 Entity *ActionBlock*. . . . 72
4.23 Entity *ExtensionPoint*. . . . 74
4.24 Entity *Flow*. . . . 75
4.25 Entity *Alternative*. . . . 77
4.26 Entities *Actor* and *System*. . . . 79
4.27 Entity *Message*. . . . 79
4.28 Entity *Ref*. . . . 80
4.29 Entity *Alt*. . . . 81
4.30 Entity *Opt*. . . . 81
4.31 Entity *Operand*. . . . 82
4.32 Entity *SystemSequenceDiagram*. . . . 82
4.33 Entity *InteractionOverviewDiagram*. . . . 84
4.34 Entity *InitialNode*. . . . 85
4.35 Entity *DecisionNode*. . . . 86

4.36 Well-formedness rule #10: Decision nodes have two or more outgoing edges. . . . 86
4.37 Well-formedness rule #11: Different outgoing edges of decision nodes must point to different nodes. . . . 86
4.38 Entity *IODRef*. . . . 87
4.39 Entity *IODSSD*. . . . 88

5.1 Transformation rule #1: Input steps to System Sequence Diagram (SSD). . . . 90
5.2 Transformation rule #2: Output steps to SSD. . . . 91
5.3 Transformation rule #3: System Responsibility (SR) steps to SSD. 91
5.4 Transformation rule #4: System Check (SC) steps to SSD. . . . 92
5.5 Transformation rule #5: Input Validation (IVAL) steps to SSD. . . 93
5.6 Transformation rule #6: User Decision (UD) steps to SSD. . . . 94
5.7 Transformation rule #7: exceptions to SSD. . . . 95
5.8 Transformation rule #8: goto next/resume flows to SSD. . . . 97
5.9 Transformation rule #9: parallel flows to SSD. . . . 98
5.10 Transformation rule #10: cyclic alternatives to SSD. . . . 99
5.11 Transformation rule #11: inclusion steps to SSD. . . . 100
5.12 Transformation rule #12: extension steps to SSD. . . . 100
5.13 Transformation rule #13: simple action blocks to Interaction Overview Diagram (IOD). . . . 102
5.14 Transformation rule #14: action blocks with alternatives to IOD. . 103
5.15 Transformation rule #15: action blocks initiated by a UD to IOD. . 104
5.16 Transformation rule #16: *Goto* steps to IOD. . . . 105
5.17 Transformation rule #17: exceptions to IOD. . . . 106
5.18 Transformation rule #18: inclusion steps to IOD. . . . 107
5.19 Transformation rule #19: extension steps to IOD. . . . 107

5.20 Textual specification of the Service Payment use case. 110

6.1 Textual specification of the Perform Session use case. 118
6.2 Transformation of Perform Session's step 1 to SSD. 119
6.3 Transformation of Perform Session's step 2 to SSD. 122
6.4 Transformation of Perform Session's steps 3 through 7 to SSD. . . . 123
6.5 Concretization of the Specialize step. 124
6.6 Transformation of Perform Session's step 8 to SSD. 125
6.7 Transformation of Perform Session's steps 9 and 10 to SSD. 125
6.8 Transformation of Perform Session's steps 11 and 12 to SSD. 127
6.9 Transformation of Perform Session's steps 13,14 and 15 to SSD . . . 127
6.10 Transformation of Perform Session's first action block to IOD. . . . 129
6.11 Transformation of Perform Session's second action block to IOD. . . 130
6.12 Transformation of Perform Session's step 8 to IOD. 130
6.13 Transformation of Perform Session's step 9 to IOD. 131
6.14 Transformation of Perform Session's third action block to IOD. . . . 132
6.15 Transformation of Perform Session's fourth action block to IOD. . . 133

A.1 Textual specification of the Transfer Money use case. 148
A.2 Textual specification of the Handle Invalid Pin use case. 149
A.3 Textual specification of the Withdraw Money use case. 150

List of Figures

1.1 The RUP “hump chart”. 2
1.2 Graphical synthesis of our contributions. 5

2.1 Example of a use case model. 9
2.2 Example of generalization between use cases. 11
2.3 Example of a textual specification of a use case. 12
2.4 General structure of action blocks. 13
2.5 Example of a system sequence diagram. 15
2.6 Example of an interaction overview diagram. 18

3.1 Alloy model of an address book. 41
3.2 Address book instance generated by the Alloy Analyzer. 44
3.3 Address book instance projected on the *Book* entity, generated by the Alloy Analyzer. 45

4.1 Graphical overview of our approach. 47
4.2 Use case meta-model. 50
4.3 General structure of action blocks. 54
4.4 System sequence diagram meta-model. 78
4.5 Interaction overview diagram meta-model. 84

5.1 The three types of alternative flows. 96

5.2 External system sequence diagrams. 102
5.3 Flow diagram of the Service Payment use case. 111
5.4 Frame schema of the Service Payment use case. 112
5.5 System sequence diagram of the Service Payment use case. 113
5.6 Interaction overview diagram of the Service Payment use case. 114

6.1 Use case diagram of an Automatic Teller Machine (ATM) system. . . 117
6.2 Flow diagram of the Perform Session use case. 120
6.3 Frame schema of the Perform Session use case. 121

A.1 System sequence diagrams of the Transfer Money and Handle Invalid PIN use cases. 151
A.2 System sequence diagram of the Withdraw Money use case. 152
A.3 Interaction overview diagram of the Transfer Money use case. 153
A.4 Interaction overview diagram of the Handle Invalid Pin use case. . . . 153
A.5 Interaction overview diagram of the Withdraw Money use case. . . . 154

List of Acronyms

API	Application Programming Interface
ATM	Automatic Teller Machine
CASE	Computer-Aided Software Engineering
CSP	Communicating Sequential Process
IOD	Interaction Overview Diagram
IS	Information Systems
IVAL	Input Validation
MDA	Model Driven Architecture
MDSE	Model Driven Software Engineering
OCL	Object Constraint Language
OMT	Object Modeling Technique
QVT	Query/View/Transformation
RUP	Rational Unified Process
SC	System Check

SPIN	Simple Promela Interpreter
SR	System Responsibility
SSD	System Sequence Diagram
UC	Use Case
UD	User Decision
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Chapter 1

Introduction

Throughout the history of software engineering, the development of computer systems has steadily been moving towards increased abstraction. From programming in assembly to more tractable languages like C, and then to object-oriented languages such as Smalltalk. Today, models form yet another abstraction layer, resulting in the so-called Model Driven Software Engineering (MDSE).

MDSE brings several advantages to software development. Since it focuses on models rather than lower level algorithms, communication between members of a team and between teams is made easier. Also, the visual nature of some models makes the design process simpler and more intuitive. The communication with stakeholders improves as well, and once there is a solid understanding of the domain, models create a reliable foundation from which to build the real system.

The Unified Modeling Language (UML) emerged in 1997 as an attempt to unify various existing object-oriented modelling languages and to provide a method to use it. Over the years UML has been augmented and detailed, but the method was left behind [Hru97]. The non-methodical use of UML naturally leads to inaccurate models and specifications that are, consequently, of little value. UML offers several different models to model different aspects of software systems. Some models

address structure while others address behaviour or architecture.

The Rational Unified Process (RUP) [Kru03] is a software development process that divides the development of a software project in four phases: inception, elaboration, construction, and transition. The work intended for each phase is performed in iterations, where the various disciplines such as requirements elicitation, implementation and testing, among others, work together to construct the deliverable intended for that iteration (Fig. 1.1). Furthermore, the RUP advocates that this process should be *use case driven*, i.e., that requirements should be captured in use cases and the rest of the development process should build upon them.

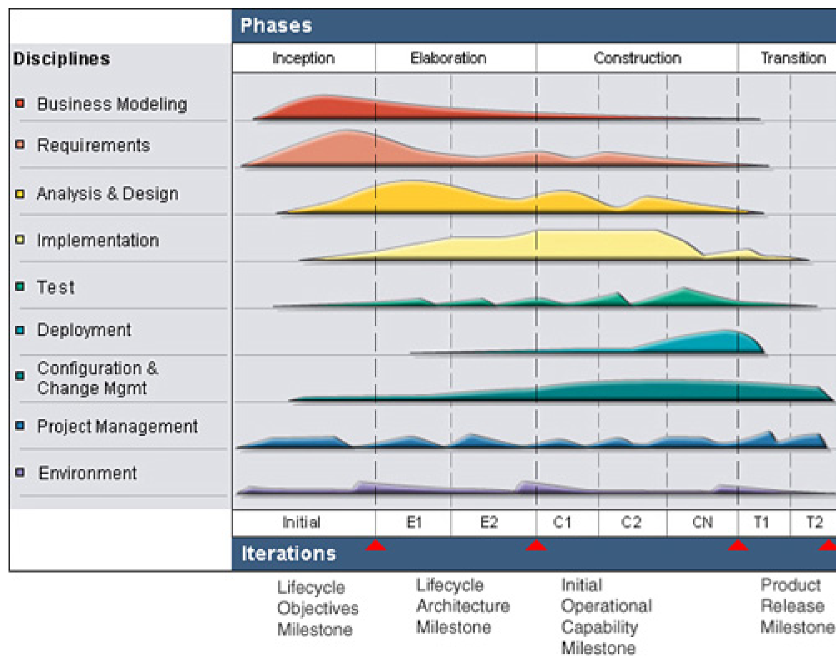


Figure 1.1: The RUP "hump chart".

1.1 Motivation

If UML models or diagrams are transformed in a non-systematic way, the situation becomes even more critical since the successive refinements will propagate

inaccuracies contained in the models, inevitably leading to wrong implementations.

Furthermore, UML documentation [OMG11] about the textual representation of use cases, the UML artefact most often used in real practice, is scarce at best. Since it is not standardized, it is possible that different modellers in the same development team have different use case writing styles, which forces subsequent development tasks to be able to interpret and adapt to each style. This is naturally error prone and makes the methodical use of UML more difficult. Even if a development team agrees to an internal standard, it is still possible that they benefit from the structuring mechanisms introduced later in this dissertation.

Another common criticism of UML is its lack of formalism. The UML specification does provide some well-formedness rules for its diagrams in Object Constraint Language (OCL) but many others are mentioned only in natural language. Besides, inter-model consistency rules are not mentioned at all. While this level of informality makes the language more flexible and allows some level of customization for different projects, it also introduces undesired ambiguity which, again, is counterproductive to the systematic and methodical use of the language.

1.2 Contributions

The method envisioned in this dissertation is also based on the RUP´s use case driven ideology but proposes a way to systematically transform use cases into other models. This allows the easy determination of the location and impact of any changes to requirements during the project and, consequently, assures traceability between use case models and other diagrams. Hence, the work developed in this dissertation fits mainly in both the inception and elaboration phases of a software project. First capturing the requirements in use cases and then using them as a solid basis to analyze and design the rest of the system.

To address the dilemma of uniformity in the use case textual representations we propose a way of identifying the individual action steps with their intention (whether they are input or outputs, etc.) and grouping them within action blocks [HA09, Hel05]. This results in a well structured and defined use case and makes it both easier to think through when writing and clearer to the reader. Furthermore, this lays down the foundation for the model transformation process.

The systematic model transformation process is made possible since some UML models, while each providing a different view of the system, have overlapping elements; i.e., elements that represent the same thing. It is based on these overlapping elements that transformation rules can be derived. In this dissertation, we propose a set of well-defined transformation rules between use cases and system sequence diagrams and interaction overview diagrams. We chose these diagrams because, while each possesses their own strengths and weaknesses, both are natural next steps after the use cases have been constructed.

While systematic, the transformation process is manual. However, in this dissertation we also propose using a formalization of the models to support the process. This allows verifying the intrinsic correctness of the models as well as verifying if a system sequence diagram or interaction overview diagram is a correct derivation of some use case model.

We use Alloy to formalize the models. Alloy is a lightweight formal modeling notation that retains the precision and expressiveness of “classical” formal languages like Z. One of the main strengths of Alloy resides in the Alloy Analyzer, which lets us visualize the model we are building while we are building it, thus promoting an iterative development. This allows modelers to easily detect and rectify defects in the model which otherwise could go unnoticed.

Fig. 1.2 synthesizes our work and contributions.

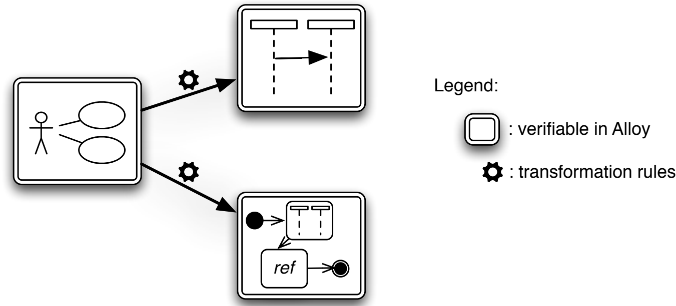


Figure 1.2: Graphical synthesis of our contributions.

1.3 Outline

The UML's problems delineated before are not new. In fact, they have been around since its genesis around fifteen years ago. Hence, there is considerable research concerning them. In Chapter 2 we give an overview of the work done in this field and clarify on how the work presented in this dissertation is different. Next, we introduce Alloy in Chapter 3. We start by explaining its logic and then the language's syntax through examples. We conclude that chapter with an introduction of the Alloy Analyzer. Afterwards, in Chapter 4, we show how we used Alloy to formalize the aforementioned diagrams and canonical textual use case representations. Chapter 5 discloses the transformation rules we created and in Chapter 6 we employ the methodology in the practical scenario of an Automatic Teller Machine (ATM) system. There, a step by step walk-through on how to correctly apply the transformations rules is given. We then conclude this dissertation with Chapter 7, where we summarize the work performed and identify some its limitations at the present stage, though suggesting possible lines of research for future work.

Chapter 2

Related Work

In this chapter, we will discuss the key concepts and previous work that are associated with our study. We begin by introducing UML and its problem of informality in Section 2.1. In Sections 2.2 and 2.3, different works that seek to address this predicament are analyzed. In Section 2.2 we discuss works that focus on the models addressed by this dissertation and in Section 2.3 we analyze various formal methods often used in the formalization of the UML, albeit applied to different models: the OCL, the Z notation, graph grammars, the Simple Promela Interpreter (SPIN) tool and Alloy.

2.1 Unified Modeling Language

The UML is a graphical modeling language that allows specifying, visualizing, constructing, and documenting software systems. It has emerged in 1997 with the aim of unifying different modeling languages existing at the time, such as the Object Modeling Technique (OMT) [RBP+91], Objectory [JCJÖ92] and the Booch Method [Boo95]. Since then, the UML has become the industry standard by, for example, facilitating communication within development teams, helping to deal

with complex systems, and allowing the development and abstraction of design patterns.

Despite numerous qualities, the UML has been criticized over the years due to some of its shortcomings [SG99, FSKdR05]. Among these shortcomings, stands out the lack of formality of its models. The specification of UML [OMG11] develops some well-formedness rules intrinsic to each model, but is mute on how to check whether different models are consistent with each other.

The UML diagrams are divided into two groups: *structural* diagrams and *behavioral* diagrams. Diagrams in each of these groups represent structural or behavioral information, respectively, in different ways and emphasizing different aspects. Although in some instances they are obvious (e.g. the methods referenced in a sequence diagram must be present in the class diagram), the UML specification is silent when it comes to clarifying this type of relationships.

In this work, we address three distinct behavioral diagrams: use case diagrams, sequence diagrams, and interaction overview diagrams. We seek to establish precise transformation rules between them and verify the rules automatically. Assuming the reader is familiar with the UML, this section provides a succinct exposition of these three diagrams placing emphasis only on the strange aspects to the UML specification.

2.1.1 Use Cases

Use cases are a technique for capturing functional requirements of a system without revealing their internal structure. Each use case represents a desired feature for a system through a sequence of messages exchanged between the system and the external entity interested in the functionality, known as *actor*, whether it is a person or a computer system [RJB04].

Use cases facilitate the communication among members of a development team

and with the stakeholders of the project. They are a core part of use case-driven methodologies, such as the RUP [Kru03], since it is from these that the rest of the development process unfolds. Additionally, use cases promote the discovery of alternative flows or exceptions to the main flow of the features required for the system, they document the development process, can be used to prioritize work, to quickly discover the gaps between the work completed and the work to be done, and if a properly systematized use case-driven methodology regarding the transformation of models is adopted, when requirements change, it is also possible to know exactly all the artifacts that need to be changed.

While the UML specification [OMG11] only alludes to the use case diagrams and does not state any way to represent the contents of use cases, these are constructed by describing the interactions that take place between the actors and the system, typically using a natural language narrative; although state diagrams, activity diagrams or interaction diagrams can be used for the same purpose [RJB04].

Use Case Diagrams

A use case is a set of sequences of actions that a system performs to provide a result of value to an actor [BRJ05]. Use case diagrams assemble use cases and show the relationship between them and the *actors* who perform them, indicating which actors perform which use cases. They also illustrate relations exclusive to actors, such as generalization, and relations exclusive to use cases, such as generalization, inclusion, and extension. When we say that generalization is exclusive to both actors and use cases we mean that an actor cannot specialize a use case and vice-versa. Fig. 2.1 presents an example of a use case diagram.

Include Relation Sometimes, distinct tasks of a system require running a piece of behavior that is common to them. Without a proper mechanism, this behavior

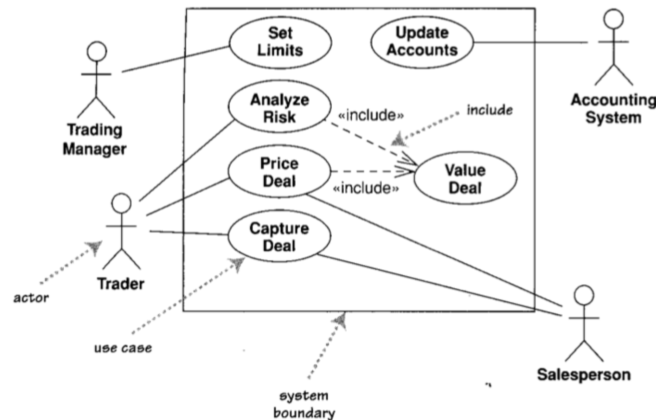


Figure 2.1: Example of a use case model [Fow03].

would have to be modeled in all use cases that wanted to run it, which could conduct to problems in consistency and maintenance.

The purpose of the include relation is, precisely, to allow the reuse of bits of behavior common to several use cases. This prevents describing the same flow multiple times. The common parts are encapsulated in a new use case that can be included in all the use cases that require the behavior. Another motivation to use include relations emerges when we intend to modularize a design, which happens when, for example, a use case proves to be too large and subdividing it provides reading and management benefits. Here, the use cases are included by just one use case.

Therefore, the include relation is a dependency between two use cases in which one incorporates the behavior defined in the other, in a point specified by the first. As such, the first, called *base use case* or *including use case*, is dependent on the second, called *included use case*, and does not make sense without it. On the other hand, an included use case does not make sense by itself, only in the context of another use case that includes it.

Thus, the include relation functions analogously to the call of a subroutine:

while running the base use case, it will reach the point of inclusion. The included use case is executed at this stage. When it completes, the base use case is resumed after the inclusion point. Note that the execution of the included use case is not optional and is necessary for the proper functioning of the base use case [OMG11].

Extends Relation Typically, a use case is not composed only by the description of the interaction in which everything runs smoothly. There are situations where, for some reason, the system cannot provide the result of value to the actor or needs to go through alternative paths to reach the end of the use case successfully. The alternative paths that a use case can take must be described in its specification as well. However, it is often necessary to provide many alternatives which are sometimes complex. When this happens, there is the risk of the use case becoming difficult to read and maintain. Therefore, the objective of the extends relation is to allow the separation of conditional behavior from the *base use case* [BRJ05].

Specifically, extends is a relation between an *extending use case* and a *base use case*, or *extended use case*, in which the extending use case may append, under certain conditions, its behavior to the base use case. Thus, the extended use case is independent of the extension use case and has meaning by itself, while the extension use case typically only makes sense within the context of the use case that extends [OMG11].

Although the extended use case makes sense by itself, under certain conditions its behavior can be augmented by another use case. The points at which the base use case can be extended are called *extension points* [BRJ05]. Associated with each point extension is at least one condition.

Whenever an instance of a use case reaches an extension point, the condition is evaluated. If the condition is found to be true, the associated extension is executed and upon its termination the base use case is resumed; otherwise, the main flow

continues normally [RJB04].

Because one can describe conditional behavior within a use case, whether to use the extends relation is at the discretion of the modeler [SP06]. In practice, the use of the extends relation proves useful when the alternative flows are complex and/or have their own alternative flows, to prevent the base use case from becoming bloated. Analogous to included use cases, extending use cases can extend different base use cases, promoting thereby the reuse of behavior.

Generalization Generalization is the most controversial relationship between use cases. The basic idea is that a child use case inherits the behavior and meaning from the parent use case, being the child able to append to or rewrite the inherited behavior while also being able to replace the parent anywhere where it appears [BRJ05]. The problem is that it has not yet been reached a consensus on how to take advantage of this concept, nor is it clear what is meant to specialize behavior [Coc00, WKKP05, Lar04]. Consequently, many authors argue that one should not use the generalization of use cases to avoid any confusion due to different interpretations of this relationship.

Anyhow, the purpose of generalization relationship is to model situations where a given task can be accomplished in different ways. Fig. 2.2 provides an example of generalization.

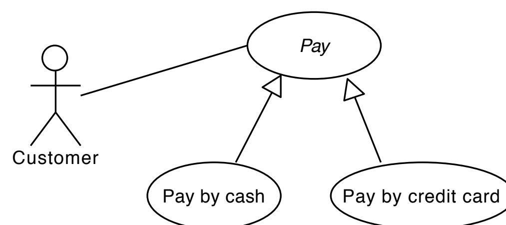


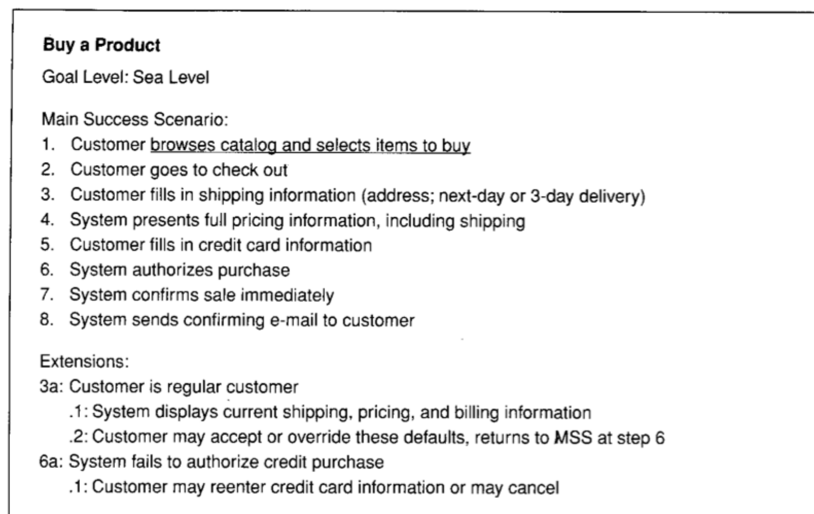
Figure 2.2: Example of generalization between use cases.

Some authors advocate that parent use cases define a base behavior which

the children may append new actions to (contiguous or not) or replace inherited actions [RJB04]. However, in the context of this dissertation, we will follow the indications given by Cockburn [Coc00], who suggests leaving the parent use case empty and specifying all the behavior in children use cases, keeping in mind that both should have the same goal.

Textual Use Cases

Textual use cases consist of the specification of a set of actions performed between actors of a system and the system itself, with the aim of providing a result of value to the actors [OMG11, Fow03]. It is these specifications that detail the actor-system dialogue of the use cases presented in the use case diagram. Common representations of these specifications include mono-column (Fig. 2.3) and dual-column narratives, where the actions performed by the actor and the system are segregated and represented in their own column.



Buy a Product

Goal Level: Sea Level

Main Success Scenario:

1. Customer browses catalog and selects items to buy
2. Customer goes to check out
3. Customer fills in shipping information (address; next-day or 3-day delivery)
4. System presents full pricing information, including shipping
5. Customer fills in credit card information
6. System authorizes purchase
7. System confirms sale immediately
8. System sends confirming e-mail to customer

Extensions:

3a: Customer is regular customer
 .1: System displays current shipping, pricing, and billing information
 .2: Customer may accept or override these defaults, returns to MSS at step 6
6a: System fails to authorize credit purchase
 .1: Customer may reenter credit card information or may cancel

Figure 2.3: Example of a textual specification of a use case [Fow03].

The UML specification [OMG11] indicates the constituent elements of use case

diagrams. However, it does not state how to organize their content, the textual specifications, or the way it relates to the diagrams. In practice, though, use cases have been specified using sequence diagrams, Petri nets or programming languages, but the most usual is the use of natural language [Coc00].

Considering that use cases serve as a means of communication between people, often without technical training, a textual format is possibly the best choice [Coc00]. Nevertheless, natural language can lead to ambiguities, making multiple interpretations possible as well as hampering their writing and understanding.

In an approach to this dilemma, Heldal [Hel05] introduced the notion of *action block*. Action blocks group *action steps*, the steps constituting the textual specifications of use cases. This grouping is based on the interpretation given in Cockburn [Coc00] of the concept of *transaction* created by Jacobson [JCJÖ92]. Specifically, action blocks are composed of: (1) an *input* made by the actor, (2) *validation* of the input, by the system, (3) a *change in the state* of the system and, finally, (4) the *response* given by the system to the actor with the result of the operation (Fig. 2.4). The input step marks the beginning of any action block, but the remaining steps can form any subset of the other steps adverted to, provided they maintain the order described.

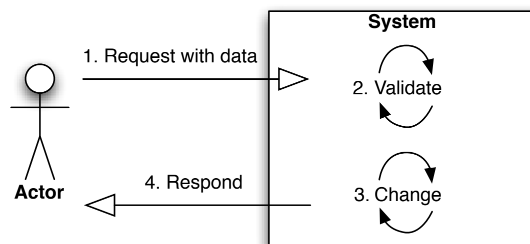


Figure 2.4: General structure of action blocks.

Further research concluded that the validation step was not flexible enough [HA09]. So, the validation step was replaced by a new step: the *assume* step.

Assume steps are not restricted to the second position in the flow of action blocks and can arise after a change of the system's state. Thus, they act not only as a validation step of the user's input but can also check the result of internal operations. Moreover, in Helldahl and Ashraf [HA09], the authors restricted the origin of alternative flows to assume action steps. In Chapter 4.1.1 we argue that the concept of assume steps can still be ameliorated and present our contributions to the development of this concept.

Another possibility for structuring use cases is to classify them according to their goal. Cockburn [Coc00] contemplates three levels for the classification of goals of use cases: *summary*, *user-goal*, and *sub-function*. Summary level use cases represent high-level business processes and involve the execution multiple instances of user-goal level use cases. These, in turn, represent tasks that provide a result of value to the actor and may depend on the execution of sub-function level use cases, which represent support operations to upper level use cases and do not constitute, by themselves, an action of value for the actor. In this work, we will only consider the user-goal and sub-function levels.

Besides classifying use cases it is also possible to classify the flows from which they are built. Each use case is composed of a *main flow* and, potentially, *alternative flows*. The main flow corresponds to the most common route that the actor-system dialogue takes to provide a result of value to the actor [SCK09]. Alternative flows may represent *exceptions* [MOW03, KG03, AM01], *alternative stories* [MOW03], *conditional insertions* [JCJÖ92, AM01, Coc00, KG03, Sim99], *cycles* [MOW03] or *alternative fragments* [MOW03].

Some authors use the term *extensions* to encompass all these types of flows. However, this term can cause confusion resulting from the extends relation provided by the UML specification. Hence, we use the term *alternatives* with the same intention.

2.1.2 System Sequence Diagrams

System sequence diagrams represent use cases graphically. They denote the actor and the system, messages exchanged between them and in which order, and the origin and destination of each message. The system, in this type of diagrams, is treated as a black box, which puts the emphasis on the actor-system interaction [Lar04].

To see the system as a black box forces us to think about the interface of the system, i.e., the things that it will be able to do. Instead of fretting, at an early stage of the software development, on how it will do them; in the same way that when we design a class, we do not think, initially, about how to implement the methods of its Application Programming Interface (API), but only to define the API. It is in this context that the usefulness of the system sequence diagram can be appreciated. To clarify the concept of system sequence diagram, one is illustrated in Fig. 2.5.

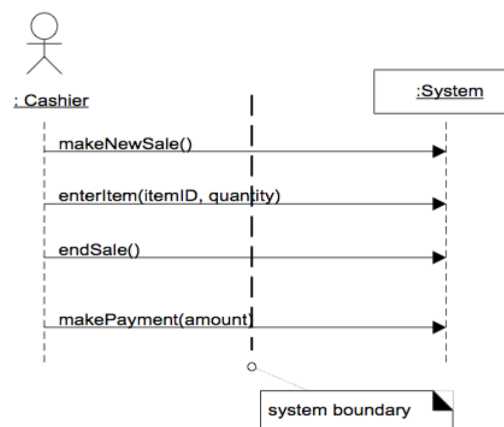


Figure 2.5: Example of a system sequence diagram [Lar04].

A system sequence diagram is contained within a rectangular frame with its name on the top left corner and is divided into two columns, designated as *lifelines*; one to represent the actor, denoted by a stick figure, and the other to represent

the system, denoted by a box with the name of the system in it. The interaction specified in use cases is, in system sequence diagrams, transformed into messages exchanged between the actor and the system. These are represented by horizontal arrows departing from the lifeline that sends the message and end in the lifeline that receives it.

System sequence diagrams support various constructs for control flow. Conditional behavior of the types *if-then* or *if-then-else*, loops and exceptions can be modeled via *combined fragments*. These fragments correspond to rectangular frames with a label indicating what type of control flow structures they represent in the upper left corner. Some fragments can be subdivided. Conditionals of the type *if-then-else*, for example, are represented in a fragment with several segments, called *operands*, divided by a dashed line.

The UML specification offers several types of combined fragments. However, in this work, we will work with only four:

alt: The purpose of this combined fragment is to model *if-then-else* structures. It may contain multiple operands, each having a guard condition. Only the one whose guard evaluates to true and executed.

break: The *break* fragment has only one operand and one guard condition. If the guard is false, the operand is ignored and execution continues normally; otherwise, the operand is executed but the rest of the diagram is not. Typically, this fragment is used for exceptional situations in which the task is aborted when some condition is met.

loop: The *loop* frame models cycles and has one operand and one guard condition. The operand is executed as long as the guard evaluates to true. Its semantics is similar to the *while* loop of programming languages like Java or C.

opt: This type of fragment represents *if-then* type structures. Therefore, it con-

tains one guard condition and one operand. Its semantics is analogous to the fragment *break* with the distinction that, if the condition is true, execution of the rest of the sequence diagram is resumed and after the execution of the operand.

2.1.3 Interaction Overview Diagrams

Albeit sequence diagrams have control flow structures, its primary purpose is to visualize how the various entities of a system cooperate to perform some task. In fact, if the flow of a task is comparatively complex, i.e., with several alternative paths, the corresponding system sequence diagram will be difficult to read due to the large number of frames that will be necessary.

A more suitable way to visualize complex flows involves the construction of interaction overview diagrams. Interaction overview diagrams mix the notation of activity diagrams with bits of sequence diagrams. The sequence diagrams illustrate linear interactions between actor and system, while the notation of activity diagrams that handles control flow deals with alternative paths and cycles.

Concretely, interaction overview diagrams are a specialization of activity diagrams in which the nodes correspond to sequence diagrams or embedded references to external sequence diagrams [GCP+05]. The execution of the diagram begins with the *initial node*, then follows the arrows and executes each node until it reaches the *final node*. The alternative paths, which in sequence diagrams are represented by the combined fragments, in the interaction overview diagram are modeled through *decision-nodes*. These contain guard conditions corresponding to each of the flows and according to the evaluation of the guard, the corresponding flow is executed. An interaction overview diagram delineating the process of online shopping is shown in Fig. 2.6 as an example.

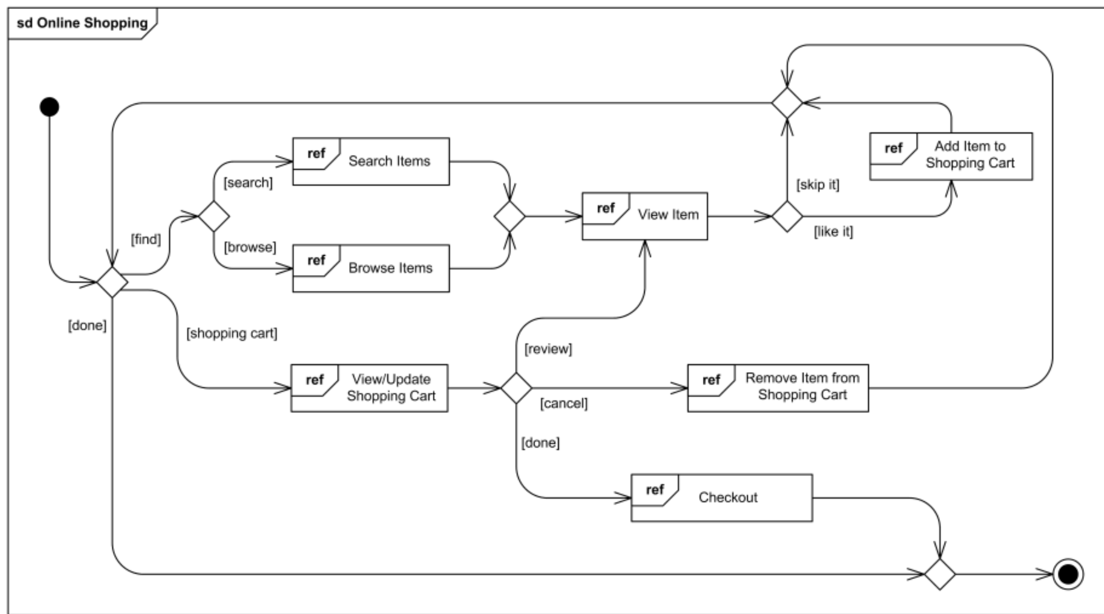


Figure 2.6: Example of an interaction overview diagram[1].

2.2 Transformation and Consistency of UML Models

One of the strengths of UML is its wide range of models that allow us to approach a software system from different perspectives. However, this is also the cause of one of its most studied problems: inconsistency.

Inconsistency problems arise when, as a software system is being developed resorting to a methodology based on model transformation, successive transformations and refinements dilute the significance of the initial models, eventually contradicting each other.

Inconsistent models may cause problems in the implementation [MBC05], in their management [KHR+03] and make it impossible to generate code through the models [SB05], a privileged process in MDSE.

[1] http://www.uml-diagrams.org/interaction-overview-diagrams-examples.html

To overcome these issues it is necessary to employ techniques to ensure the consistency of UML models. This need becomes apparent if we analyze the amount of published work in this area in recent years, whether in journals, workshops [KHRS02, KHR+03, KHR+04] , or conferences. Thus, model transformation and consistency are two closely related topics [EB04]. If, on one hand, without a notion of consistency, the models can be freely transformed, the lack of transformation rules between two models (for lack common concepts), means that it makes no sense to speak of consistency between them.

2.2.1 General Terms

To better understand the area under study, it is important to clarify some of its key concepts such as *model*, *model transformation*, *transformation rules*, *intra-model consistency* and *inter-model consistency*.

Model A model is an abstraction of something, in which important aspects of the object being modeled are captured and aspects seen as less relevant are omitted or simplified. Software systems are modeled through modeling languages such as UML, where they are represented by diagrams and text. The model should be able to answer questions instead of the real system and be easier to use [BG01, RJB04].

Model Transformation Model transformation involves the construction of a *target model* from a *source model* according to a set of transformation rules. This definition is akin to that given in Kleppe et al. , e[KWB03]xcept the transformation in that work is considered an automatic process, whereas in this work the transformation will be performed manually.

Transformation Rule A transformation rule is a description of how the source model's constructs are mapped to the constructs of the target model [KWB03].

Intra-Model Consistency Property of a model when it conforms to its metamodel and respects all the rules imposed on it [EB04].

Inter-Model Consistency Property observed between a source model and a target model when, by applying the transformation rules corresponding to the two models, from the source model it is possible to derive the target model [EB04].

2.2.2 Consistency Approaches

Because the literature about the UML consistency is quite extensive, it is possible to find many different approaches to this problem, formal and informal. Within the group of approaches that use formal techniques there are authors who make use of, for example, graphs, petri nets, Z, B, Communicating Sequential Process (CSP) or description logics.

In this work, we will consider transformations of use cases to sequence diagrams and use cases to interaction overview diagrams. So, in this section, we will pay special attention to literature that focuses on these specific diagrams, leaving to Section 2.3 a discussion of different approaches concerning the formalization of UML which work with other diagrams.

One possible way to accomplish the transformation between use cases and sequence diagrams is based on natural language processing [YBL10, SHH07, Li00]. Yue et al. [YBL10] presents an approach and a tool that perform the automatic generation of sequence diagrams from use cases. However, the transformation is also based on class diagrams and the operations depicted therein. In addition, generated sequence diagrams are not system sequence diagrams and the writing of textual use cases is constrained with the use of keywords (e.g. *if*, *then*, *else*, *meanwhile*) to define the flow of the use case. The tool presented in Segundo et al. [SHH07] also imposes rules on how the use case should be specified to

be able to do the language processing. It accepts only specifications in Spanish and its grammar rules, not referring under which circumstances the characteristic sequence diagrams' frames are used or even if the tool supports them. Li [Li00] resorts to the use of keywords to write the flow of use cases as well. The presented approach is semiautomatic, requiring user input to translate certain parts of the use cases. Like Second et al., it makes no mention of the use of frames.

Furthermore, the application of approaches making use of natural language processing is inherently limited to environments in which the language for which they were developed is used, thus hampering widespread adoption.

Apart from approaches based on natural language processing, other solutions were sought by different authors. In Almendros-Jimenez and Iribarne [AJI07] an approach based on several steps was carried out: first, a use case diagram representing the basic tasks of the system is developed; then, the behavior of use cases is detailed resorting to sequence diagrams and encapsulating some bits of behavior in sequence sub-diagrams; and finally, these sequence sub-diagrams are mirrored in the use case diagram where they represent the inclusion and extension of use cases. However, this approach does not use textual specification of use cases, important artifacts to communicate with stakeholders. In Mason and Supsrisupachai [MS09], it is presented a tool where one can write the textual specifications of use cases and label certain words as being the *sender*, *receiver*, or the *message*, including parameters. This approach bypasses the linguistic limitation and allows the unrestrained description of use cases but the cost of an intrusive technique that impairs the readability of use cases. In addition, this paper does not mention any of the characteristic sequence diagrams' frames as well.

As far as the transformation from use cases to interaction overview diagrams goes, literature is rather scarcer. The interaction overview diagram is a new type of diagram that emerged only with the UML 2.0 and its optimal use is still to be

understood. Nevertheless, Garcia et al. [GCP+05] presents a use case diagram and the corresponding interaction overview diagram in different phases; i.e., first considering simple diagrams without inclusion or extension relationships and then elaborating on that by incorporating them. However, no kind of systematic rules to perform this transformation is indicated.

Regardless, before one can talk about inter-model consistency, it is necessary that the individual models are in accordance with their well-formedness rules. The starting point for the definition of well-formedness rules is the UML specification [OMG11]. Therein some rules are provided by using OCL [OMG10] or natural language for each diagram. The use of natural language to define the semantics of the diagrams provides greater flexibility for modelers because they can be interpreted in different ways and allow the application of the diagrams in completely different contexts. However, when attempting to formalize the rules, this form of presenting them is counterproductive.

Consequently, some authors addressed this question by using more rigorous methods to formalize the rules. In Ibrahim et al. [IIS+10] the authors defined and formalized well-formedness rules for use case diagrams using set theory, but did not indicate or use any mechanism for the automatic verification of their correctness. In contrast, in Ballur and Vallieswaran [BV06], it is presented a tool for checking the consistency between design and code where some well-formedness rules are presented for use case diagrams and for system sequence diagrams, among others. However, it does not address the extends, include or generalization relations.

2.3 Formal Methods

"*The nice thing about graphical description techniques is that every one understands them, the bad thing, however is that every one understands them in a dif-*

ferent way." This is the problem of notations such as UML and where the application of formal methods can bring significant advantages, by making explicit and unequivocal the semantics of the notation.

Formal methods is a set of mathematical languages, tools and techniques for the specification and verification of software and hardware systems. It is particularly useful in critical and/or large scale systems in which a small error can cause great losses of money, time, or even human lives. Albeit formal methods cannot guarantee 100% correction of the system, its aid in the discovery of inconsistencies, ambiguities and incompleteness, allows for the elaboration of models with greater reliability and robustness [CW96].

In this section, we will review the formal methods that are most often applied in the formalization of UML. We will highlight some of the characteristics of each approach and conclude with the justification for choosing Alloy as our model checking language in this work.

2.3.1 OCL

OCL [OMG10] emerged along with UML as the formal language to describe expressions on its models. Typically, these expressions represent invariants of the model or queries that can be performed on it, bearing in mind that the evaluation of those does not produce any side effects, i.e., it is not possible to change the state of a system through OCL.

Chiorean et al.[CPC+04] uses the XML Metadata Interchange (XMI) standard for transferring consistent models from the Object Constraint Language Environment[2] checking tool to another UML tool, and vice versa. Besides detecting inconsistencies, the tool is also able to correct them. As in this dissertation, the consistency rules are defined at the meta-model level, making them independent

[2] http://lci.cs.ubbcluj.ro/ocle/index.htm

from the user's models. However, the authors expound only the well-formedness rules and do not allude to issues of inter-model consistency.

On the other hand, in Bodeveix et al. [BMP+02], consistency rules between models are established. The authors discuss the consistency between class diagrams and sequence diagrams, class diagrams and object diagrams, class diagrams and state diagrams, and between sequence diagrams and state diagrams; but ignore interaction overview diagrams and use cases, the core diagram in this dissertation.

2.3.2 SPIN model checker

Simple Promela Interpreter (SPIN) [Hol04] is an open-source model checking tool and is particularly suited for concurrent systems. As the acronym suggests, the models are specified using the Promela language and are also executable. The model checker allows the simulation and exhaustive analysis of the behaviors specified.

The SPIN model checker is used in Zhao et al. [ZLQ06] to check the consistency between sequence diagrams and state diagrams. The state diagrams are mapped to the formalism of *split automata* to deal with the hierarchical states characteristic of these diagrams, which contributes to the efficient passage of these diagrams to SPIN since their semantics are analogous to that of processes in Promela. In addition, they can prove that the translation of UML models to the model checker is correct and that the model translated truly represents the UML model.

2.3.3 Z

Z notation [WD96] is a language based on mathematical set theory and mathematical logic. The combination of the Z notation with natural language allows the creation of formal specifications, which can be reasoned with by using proof

techniques of mathematical logic. These specifications can be refined successively, getting closer to final code at each iteration.

Amalio et al. [ASP04] outlines an approach to formal analysis of UML models through a formal representation in Z. The authors advocate the use of modeling frameworks for the construction and analysis of domain-specific models. Each framework consists of several UML diagrams whose semantics is made explicit in any formal specification language and where the analysis is performed using the methods available for the chosen language. Thus, the authors intend to make development in UML more precise, avoiding the need for modelers to have a thorough cognition of formal languages and thereby making the development of sound systems more practical. As an example of their methodology, the authors use Z as the formal specification language for describing the semantics of class diagrams, objects and states.

2.3.4 Graph Transformation

Graphs are a widely used structure in software engineering by allowing to explain complex problems in a natural way. At the heart of graph transformation is the rule-based modification of graphs. Each rule consists of a pair of graphs, L and R. Applying this rule to a graph consists of finding on this graph a sub-graph corresponding to L and replacing it with R [EEPT10]. The nature of the graph naturally lends itself to class and object diagrams and to the representation of system states, important artifacts when modeling in UML. Consequently, with the advent of model transformation in model driven development processes, graph transformation appears as a natural choice for its formalization.

Wagner et al. [WGN03] presents a flexible and incremental consistency management framework for the open-source Computer-Aided Software Engineering (CASE) tool Fujaba. The framework allows the user to specify consistency rules

through a formalism based on graph grammars, which allows customization of rules for different projects or domains. However, only model well-formedness issues are mentioned and not their inter-model consistency.

On the other hand, an attempt to integrate different UML models was conducted in Kuske et al. [GKZ09]. The authors used the formalism of graph transformation to provide an integrated semantic base for these. Specifically, the authors considered class diagrams, sequence diagrams, object diagrams, state diagrams, and collaboration diagrams. Despite making an effort to integrate more diagrams than other works of the same genre, similarly to those, use case diagrams and interaction overview diagrams are left out.

2.3.5 Alloy

Alloy is a formal language based on OCL and Z, combining OCL's emphasis on binary relations and the navigational way of expressing constraints with Z's simpler semantics [Jac00]. It retains the characteristics of formal notations such as precision and expressiveness, but adopts an automatic mechanism for the verification of specifications instead of theorem proving, typical of other formal approaches. Alloy is discussed in more detail in Section 3.

Using Alloy, in Nimiya et al. [NYM+10], the authors address the consistency between state diagrams and communication diagrams. They use a methodology analogous to the one used in this dissertation, starting by defining the models' entities and then building an instance, using assertions and predicates to validate it. Given the similarities, this article is a probable reference for potential extensions to the work of this dissertation if state and communication diagrams were to be addressed.

Class diagrams enriched with OCL constraints and corresponding transformation to Alloy are discussed in Anastasakis et al. (2010) [ABGR10]. The transfor-

mation is based on Model Driven Architecture (MDA) techniques and is intended to allow the analysis of class diagrams. The paper also addresses the challenges inherent to this transformation which arise due to fundamental discrepancies from Alloy's and UML's design, such as the lack of support for multiple inheritance in Alloy.

Shah et al. [SAB09] performed a study involving the UML2Alloy[3] tool. This tool performs the transformation of class diagrams enriched with OCL constraints to Alloy, to be able to analyze such models. In this work the authors present a new transformation that translates instances generated by Alloy into UML object diagrams, implementing the technique through the Query/View/Transformation (QVT) standard.

Barajas [Bar06] develops a formal specification for a tool that models system requirements through use cases. Specifically, a possible use case modeling in Alloy is presented, akin to that sought in this dissertation. Some of the definitions presented in the study were reused here, while others were discarded because of different interpretations regarding the semantics of relations between actors and use cases.

In Kelsen and Ma [KM08], the authors argue that Alloy is a good approach to formalizing modeling languages. The study compares several alternatives and concludes that the greatest advantages of Alloy are a uniform notation and automated analysis. These are the reasons that guided us to electing Alloy for the development of the consistency mechanism presented in this dissertation.

[3] http://www.cs.bham.ac.uk/~bxb/UML2Alloy/

2.4 Conclusion

In this chapter, we explained the value of the application of formal methods as complement to less formal modeling languages like UML. They consist of a set of languages, tools and mathematical techniques which are particularly suitable for critical or large scale systems' design. We discussed several formal methods, such as OCL, the Z notation, the SPIN tool, graph grammars and Alloy, which are often used in UML formalization and how studies using these methods were related to the work proposed in this dissertation. We found that albeit the literature is extensive, relatively few works address the formalization of use cases and interaction overview diagrams. This justifies the need for a research as the one developed in this dissertation. Finally, we selected Alloy as our modeling language because of its simple notation and powerful automatic analyzer.

Chapter 3

Alloy

Despite the advantages of formal methods, its application in the industry has been limited [Kne97]. This has conducted to the emergence of formal methods whose emphasis is on the partial and focused application of the formalization, the so-called lightweight formal methods [JW96].

Alloy [Jac12] fits into this new type of formal methods but retains some of the qualities recognized to traditional formal methods such as precision and expressiveness. The main distinction between these two kinds of formal methods consists in the use of a mechanism for automatic analysis instead of analysis based on theorem proving. The disadvantage of this mechanism is its limited space of examined cases, while the theorem proving provides a valid response to any element within the spectrum of possible cases (which may be infinite). Nevertheless, the number of cases analyzed is in the order of billions and thus considerably higher than the degree of coverage achieved with testing [Jac06].

The models specified in Alloy are typically small (*micromodels*), *analyzable*, *declarative*, and *structural* [Jac02].

They are typically small because it is possible to study properties of a complex system with few lines of code, the language itself is simple and small, and the focus

is on formalizing only the parts of a system that really should be formalized, either because they present substantial complexity or because they support processes whose failure could cause considerable losses.

Alloy allows its models to be analyzed in two ways: simulation and verification. Simulation is based solely on the model and generates possible instances, allowing us to conclude whether the model is consistent or not. Verification requires as input some property that the modeler wants to check. If the model does not possess that property, Alloy produces a counter-example; i.e., an instance of the model where the property does not hold.

The Alloy language is also declarative. Models constructed using this paradigm allow the modeler to focus on communicating *what* happens when some operation is performed instead of worrying about *how* it happens. In state-based languages like Java or C, the programmer explicits the way to reach some state from another, while in declarative languages (like Prolog or Haskell) the focus is on stating how the initial and final states are connected.

Software systems are composed of two fundamental elements: structure and behaviour. There are several analysis tools for the behavioural part; however, Alloy is one of the first to approach the structural component. Alloy allows the definition of complex structural relations between the various entities belonging to the problem's domain so that one may then reason about their properties.

Unlike UML, which is a visual modelling language, Alloy is mainly text-based. Alloy models are composed of *signatures*, *facts*, *predicates*, *assertions*, *functions* and *commands*. Domain entities are introduced in Alloy models via the signature construct. Each signature is allowed to have a set of properties, effectively relating them with other signatures. Constraints can be applied to these relations to restrain the way signatures may associate with each other. In Alloy, these constraints are introduced via facts, which represent properties that the model must

possess; or via predicates, for when we want to analyze a model with and without a certain property. Predicates also allow, like functions, to package expressions for reuse in different contexts. The difference is that predicates evaluate to either true or false while functions return a value of a type defined by the modeler. Assertions allow us to verify if some property follows from the facts of the model and the commands are used to analyze the model.

In the remainder of this chapter, the core aspects of Alloy will be explained in greater detail. Specifically, its underlying logic and the relational perspective with which one can read Alloy models (Section 3.1), the other two perspectives, i.e., object-oriented and set theoretic (Section 3.2), the Alloy language (Section 3.3) and its analysis mechanism (Section 3.4).

3.1 Atoms and Relations

The structural component of a software system consists of the entities belonging to the problem's domain and the way they relate to each other. In Alloy, these correspond to *atoms* and *relations*, respectively.

Atoms are characterized for being *indivisible*, *immutable*, and *uninterpreted*. The fact that atoms are indivisible means that they cannot represent, for example, tuples, as they are divisible. The immutability property signifies that it is not possible to alter the properties of an atom, and being uninterpreted means that atoms do not possess any properties by default, contrary to numbers, for example.

To model divisible, mutable, or interpreted concepts, relations are used. Relations associate atoms with one another, creating tuples which may be of any arity equal to or greater than one. The arrangement of atoms in each tuple is relevant and unary relations (i.e., with arity equal to one) simply represent *sets* of atoms. A unary relation composed of a single atom is a *scalar*.

Relations can be manipulated via *set operators* and *relational operators.* The former are utilized when it is intended to work with the relations as if they were just sets of tuples. On these occasions, the structure of the relations' tuples is irrelevant and may be seen as sets of atoms. On the other hand, the structure of the tuples is essential to the latter operators, which take full advantage of the power of relations.

Set operators are based on those of set theory, namely: *union* (`+`), *intersection* (`&`), *difference* (`-`), *subset* (`in`) and *equality* (`=`). Though the structure of the tuples in this type of operations is irrelevant, these operators can only be used between relations with the same arity.

Relational operators are based on relational logic and they are responsible for making relations such a powerful mechanism. Alloy possesses the following relational operators: *product* (`->`), *dot join* (`.`), *box join* (`[]`), *transpose* (`~`), *transitive closure* (`^`), *reflexive-transitive closure* (`*`), *domain restriction* (`<:`), *range restriction* (`:>`) and *override* (`++`).

There is still a third group of operators: *logical operators.* These operators are used to combine expressions to form more complex restrictions. They are identical to those of boolean logic and each one has two possible representations: *negation* (`not`, `!`), *conjunction* (`and`, `&&`), *disjunction* (`or`, ||), *implication* (`implies`, `=>`), *alternative* (`else`, `,`) and *bi-implication* (`iff`, `<=>`).

Besides operators, Alloy also introduces some *constants.* The constant `univ` represents the set containing every atom of the model, `none` is the constant which does not contain any atom and the `iden` denotes the binary relation where each atom is related to itself.

3.2 Object-Oriented and Set Theoretic Views

Although atoms and relations reflect the true semantics of the Alloy language, when reading Alloy models one tends to think in a more abstract manner. Namely, Alloy code can be read as if it were object-oriented, which is the highest level of abstraction at which one can read Alloy code, or one can think in set theoretic terms. Actually thinking of Alloy models as atoms and relations is not usually done even by advanced users.

Entities defined in Alloy models can be seen as being classes, and the relations defined in such entities may be seen as being the classes' fields. Also, the dot join operator allows producing code similar to that of object-oriented languages, which further contributes to reading models in this manner. Interpreting Alloy models in this object-oriented fashion aids in rapidly understanding a model, but may be dangerous without a deeper understanding of Alloy, as its level of abstraction may fail to explain some of the intricacies of the Alloy language.

Thus, it is usual to think of Alloy models in set theoretic terms. Here, the entity definitions in Alloy models are regarded as sets, and individual entities (or instances, in object-oriented terms) are their elements. The relations defined in the entities are now read as mapping the corresponding entity to other entities of the type defined in the relation, which constitutes a lower abstraction level of thinking about Alloy models but which is more in line with its true semantics.

3.3 Language

In this section, we present the main characteristics of the Alloy modeling language. The syntactical constructions considered most relevant will be presented through small examples. Concretely, we will see how to introduce *signatures*, *facts*, *functions*, *predicates* and *assertions* in Alloy models, as well as how to analyze them

via the invocation of predicates and assertions.

3.3.1 Signatures

The aim of signatures is to model the entities that exist in the problem's domain, and they represent sets of atoms (or classes, in an object-oriented view). For example, the declaration

```
sig A {}
```

introduces the set *A* in the model.

Similarly to object-oriented programming languages, Alloy also supports inheritance mechanisms

```
sig A1 extends A {}
sig A2 extends A {}
```

The signatures *A1* and *A2* are both subsets of *A* in which the elements of one subset are not present in the other; i.e., the signatures *A1* and *A2* are mutually exclusive.

It is also possible to create abstract signatures by using the keyword *abstract*:

```
abstract sig B {}
```

Abstract signatures contain no elements by themselves. Thus, they are of value to a system only when they are *extended*, at which point they encompass all the elements belonging to their *subsignatures*.

Alloy further allows the restriction of the number of elements a signature may have, sufficing to precede the keyword `sig` by a *multiplicity* keyword. For example, the declaration

```
one sig C {}
```

indicates that signature C shall contain exactly one element. Besides `one`, the possible multiplicities are `lone`, `set` and `some`. These constrain the number of elements of a given signature to one at most, any number, or at least one, respectively. In case of omission, the default multiplicity is used, i.e., `set`.

Entity properties are declared as fields in the signatures as follows:

```
sig D { p : e }
```

Property p represents a relation with domain D and codomain given by the expression e. Preceding e with some multiplicity constraint it is possible to limit the number of elements that may be part of p's range. Here, the default multiplicity is `one`.

Now that we have introduced signatures we are in position to give a small Alloy example to illustrate the multiple ways to read Alloy models. Consider the following snippet:

```
sig S extends E { F: one T }
fact { all s:S | s.F in X }
```

Facts will be explained later but right now its meaning is not important nor necessary to understand the distinct ways one can read Alloy models. Thinking in an object-oriented perspective, this Alloy fragment can be read as:

- S and E are classes
- S is a subclass of E
- S has a field or attribute of type T
- s is an instance of S
- . accesses a field

- s.F returns something of type T

Interpreting the model in a set theoretic perspective, i.e., thinking about sets, elements, and relations among them, is a relatively safer approach since it does not lead to errors that the object-oriented approach might:

- S and E are sets
- S is a subset of E
- F is a relation which maps each S to exactly on T
- s is an element of S
- . composes relations
- s.F composes the unary relation s with the binary relation F, resulting in a unary relation of type T

The lowest level of abstraction, atoms and relations, is rarely used to reason about Alloy models, even though it corresponds to Alloy true semantics:

- S and E are atoms
- the containment relation maps E to S
- F is a relation from S to T
- the containment relation maps S to s
- . composes relations
- s.F composes the unary relation s with the binary relations F, resulting in a unary relation t, such that the containment relation maps T to t

3.3.2 Constraints

There are two ways to introduce constraints on a model: using *facts* or *predicates*. The difference between them lies in the fact that constraints introduced through facts are always applied to the model, while constraints introduced through predicates are applied only when they are invoked. *Assertions* allow us to verify if the model contains, implicitly, a property as a result of the constraints applied explicitly. Analogously to predicates, it is necessary to invoke an assertion to observe its effect. *Functions* represent reusable expressions that can be useful, like predicates, in various contexts. To see how to invoke predicates and assertions see Section 3.3.3.

Facts

Facts are introduced in Alloy models via the `fact` keyword. For example:

```
fact { all a: A | C }
```

introduces a constraint C for all elements of signature A. These type of facts, where a constraint is applied to all elements of a signature, can be written more succinctly through *signature facts*. Using these, the previous example would now be:

```
sig A {} { C }
```

Constraints written this way are implicitly quantified over the elements of the signature. Concretely, the previous example is the same as having:

```
sig A {}
fact { all this: A | C }
```

Besides this implicit quantification, the expansion of references to the associated signature's fields is also done implicitly. For example, consider a graph with *nodes* and *connections* between them where the nodes cannot be connected to themselves:

```
sig Node {}
sig Connection { predecessor, successor: Node } {
    predecessor != successor
}
```

implicitly, this is equivalent to

```
sig Node {}
sig Connection { predecessor, successor: Node }
fact { all this: Connection |
    this.predecessor != this.successor
}
```

However, this expansion is sometimes undesirable. To stop a field from being implicitly expanded we can use the symbol @. Considering now that we want to establish that for each connection between two nodes there is some connection in the opposite direction we could write:

```
sig Node {}
sig Connection { predecessor, successor: Node } {
    some c: Connection |
        c.@predecessor = successor and
        c.@successor = predecessor
}
```

This way, only the right side of the equalities would be expanded, which would result in the following equivalent code:

```
sig Node {}
sig Connection { predecessor, successor: Node }
fact { all this: Connection |
            some c: Connection |
                c.predecessor = this.successor and
                c.successor = this.predecessor
}
```

Predicates

Predicates encapsulate constraints which shall be applied only when invoked. This behaviour is useful, for example, when one wants to verify a model with and without a certain property. The fact that the constraint is encapsulated also allows it to be applied in different contexts or at different places in the same model, thereby promoting reuse. Although the properties described in predicates are applied only when the predicate is invoked, predicates may be invoked implicitly if they are included in the body of a fact.

Predicates are introduced using the `pred` keyword. They are parameterizable and return a boolean value. For instance:

```
pred pred_name [ p1: Param1, p2: Param2, ... ] { ... }
```

Assertions

Assertions represent constraints or properties, with the goal of verifying if these are a logical consequence of the facts of the model. If when an assertion, representing a property that we believe to be implicit in the model due to its facts, is executed and proves to be invalid, then that means that the model contains a flaw (assuming the assertion is well coded). After correcting of the flaw, it may make sense to keep the assertion in the model to make sure that future changes do not break the

property. Thus, assertions are useful for detecting errors in the models and may function as regression testing. In Section 3.4 it is possible to see how the analysis of an assertion is performed.

To declare assertions, the keyword `assert` is used:

```
assert assertion_name { ... }
```

Naming the assertion is optional, however, the command which checks assertions needs it. Thus, anonymous assertions are seldom used.

Functions

Functions are similar to predicates. However, instead of returning a boolean value, their return value conforms to the codomain defined in the function's signature. The keyword `fun` introduces functions:

```
fun fun_name [ p1: Param1, p2: Param2, ... ] : e { ... }
```

p1 e *p2* represent arguments that a function may have and *e* represents the codomain expression.

Alloy Model Example

Having presented the syntax required to specify Alloy models, we now show a small sample model (Fig. 3.1). The model's domain consists of an address book where each entry maps a person's name to another name or her address. Reasons for mapping a name to another name include being able to store a person's nickname alongside its name or to indicate that some person has the same address as another one. Whatever the case, the goal is to be able to find any person's address. For this purpose, we introduce, in line 5, the fact *invariant* which characterizes the address book's well-formedness properties.

```
1  sig Addr, Name { }
2  sig Book {
3    addr: Name →(Name + Addr)
4  }
5  fact invariant {
6       all b: Book, n: Name | some (b·addr)·n ⇒ some n·(b·addr)
7      all b:Book, n: Name | n not in n·^(b·addr)
8  }
9  assert fromInvariant {
10     all b: Book, n: Name | some (b·addr)·n ⇒ some n·^(b·addr) & Addr
11 }
12 pred singleMapping {
13   all b: Book, n: Name | one b·addr[n]
14 }
15 fun lookup [b: Book, n: Name] : set Addr {
16   n·^(b·addr) & Addr
17 }
```

Figure 3.1: Alloy model of an address book.

First, we need ensure that for every name that appears on the right side of the *addr* relation, there is an entry in the book where that name is on the left side. This is assured by the constraint depicted in line 6.

Then, we have to eliminate the possibility of a name A referring to a name B and B referring back to A. This kind of cyclic referral is tackled by the constraint on line 7, where we state that a name n cannot be an element of the set that aggregates the names and addresses obtained by recursively following the *addr* relation starting from n.

Having defined these two constraints, we introduce an assertion into the model to help ascertain that they are indeed enough to enforce the initial goal of being able to find any person's address. In the assertion, named *fromInvariant*, we intend to verify that if a name n appears on the right side of the *addr* relation, then there is some address in the set consisting of the names and addresses obtained

by recursively following the *addr* relation starting from n, the same set that we previously impeded of containing the name n.

The current model allows a name to be mapped to multiple addresses and other names. However, we might want to analyze the subset of address books where each name can be mapped only to one address or one other name. This can be done by using predicates, and the predicate *singleMapping*, on line 12, introduces a constraint that states exactly that.

Finally, on line 15, we introduce the *lookup* function which, given an address book and a person's name, returns the set of addresses associated with that person.

3.3.3 Commands and Scope

To analyze a model it is necessary to code a *command* and tell Alloy to execute it. Associated to the command, we can indicate the scope for each signature, i.e., the number of instances we want each signature to have. By default, each top-level signature, i.e., that is not an extension of another signature, will contain at most three elements.

There are two distinct commands one can use, `run` and `check`, each corresponding to a different philosophy. The former executes predicates and the latter checks assertions. When a `run` command is executed, the tool searches for instances of the model that, besides all of its constraints, also respects the property enclosed in the predicate. If one wants just to verify if the model is consistent, it is possible to apply the `run` command to an empty predicate. The `check` command, unlike `run`, searches for instances that respect all constraints defined in the model plus the *negation* of the expression described in the body of the assertion; this is useful, for instance, when one wants to check if some property follows implicitly from the constraints applied in the model.

3.4 Alloy Analyzer

The Alloy Analyzer is Alloy's component responsible for analyzing the models. It translates the specifications to boolean formulas which are then interpreted by a SAT (boolean satisfiability) solver embedded in the tool. If the SAT solver does not find any solution, the Alloy Analyzer simply lets the user know that no instance or counter-example was found. Otherwise, the resulting boolean formula is translated back to a relational formula, representative of an instance.

The resulting instance may afterwards be visualized in the tool. Through the visualization of the instances derived from the model subtle errors become apparent, errors that would hardly be detected otherwise. This makes the analyzer a tool of great practical value.

Alloy's logic is undecidable; i.e., it is not possible to be certain that an assertion is valid for all instances of a given model. To solve this dilemma a compromise had to be made: instead of trying to build a proof that the assertion is correct (similarly to theorem provers), the analyzer tries to find an instance that *refutes* the assertion. The search for the counter-example is carried out in the set of possible instances for the model. If a counter-example is found, then the assertion is invalid. However, if no counter-example is found, that does not mean the assertion is valid. It is possible that an instance that violates the assertion exists in a set bigger than the one considered.

What is, then, the factor that limits the size of the analyzer's search space? The *scope*, which was mentioned in the previous section (3.3.3). By limiting the scope of every signature, i.e., the number of elements that each signature may have, one limits the solution space of a model. Inside the space, search for the counter-example is exhaustive; i.e., all hypotheses are contemplated.

Although the search space is limited, Daniel Jackson [Jac06] believes in the "*small scope hypothesis*" which states that most bugs have small counter-examples.

In other words, if a counter-example to some assertion exists, most probably it is revealed in small scopes.

To give an example of the output of the Alloy Analyzer let us consider the address book model introduced earlier (Fig. 3.1). To analyze the model applying the constraint defined in the *singleMapping* predicate one could use the following command:

```
run singleMapping for 3 but exactly 1 Book
```

Note that we specify we want the analyzer to consider only instances where there are at most three atoms for each signature, except for the *Book* entity, for which we want exactly one atom; this is our scope. A possible instance respecting all the constraints stated in the model plus the scope we just defined is depicted in Fig. 3.2.

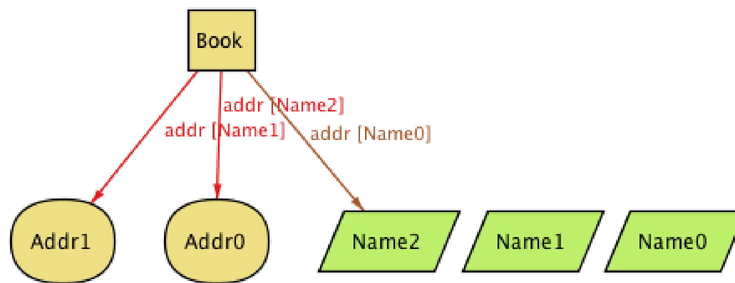


Figure 3.2: Address book instance generated by the Alloy Analyzer.

At a first glance, it may appear that atoms *Name0* and *Name1* are not contained in the address book. A closer look, however, reveals that the *Book* atom is related to *Addr1* via *Name1*, and related to *Name2* via *Name0*. Nonetheless, this view does not show clearly the organization of the address book. This is due to there not being an intuitive way to represent n-ary relations such as *addr*. To improve readability it is possible to *project* the instances on chosen signatures. In this case, projecting out *Book* results in a much clearer instance where the contents of the address book are easily understood (Fig. 3.3).

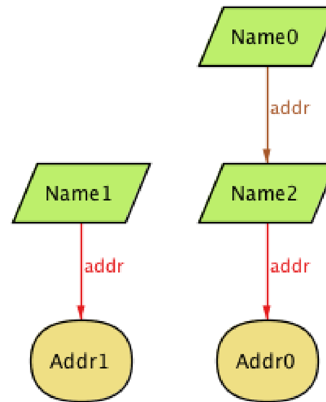


Figure 3.3: Address book instance projected on the *Book* entity, generated by the Alloy Analyzer.

3.5 Conclusion

In this chapter we have seen where Alloy fits in the area of formal methods and we introduced the defining characteristics of its models, which are *micromodels*, *analyzable*, *declarative*, and *structural*. We saw that structures in Alloy are represented by relations and presented the operators that let us express properties for those structures. Besides examples of the syntax used to construct Alloy models, we tackled the way these are analyzed via the Alloy Analyzer and the relevance of the concept of scope.

Chapter 4

Alloy Models of UML Diagrams

For the development of our approach to software modelling we explored three distinct UML diagrams: use case, sequence, and interaction overview diagrams. Use cases are used to capture the system's requirements and, therefore, are constructed at an early stage of the software development, especially when the development methodology is use case driven, like ours is. Sequence diagrams, and particularly system sequence diagrams, are well suited to provide a visual focus of the processes detailed in textual use cases, while keeping the same abstraction level and laying down the basis for posterior refinement. In contrast, the focus of interaction overview diagrams is on supplying a higher level view of textual use cases, where internal atomic processes are abstracted and encapsulated; thus granting a better understanding of the overall process by concealing nonessential details.

Modelling these diagrams in Alloy enables us to verify their well-formedness. We based our modelling efforts on the meta-models provided by the UML specification, but adapted them to better suit our needs. Particularly, we expunged some details and incorporated others where we deemed reasonable. This approach, however, is not applicable only to these two diagrams and may be put to good use should future extensions of this work aim to use different models.

However, the UML documents do not yield any meta-model for textual use cases; so, there is no standard way to specify them. To address this void, we refined and combined previous research, contributed by separate authors, on use case structuring mechanisms. The concepts of action steps and action blocks, in particular, are pivotal to our canonical textual specifications, which allows us to normalize the way use cases are written. With these concepts and structures consolidated and well defined, it was possible to integrate them with the already existent Alloy meta-model for use case diagrams. This is also what makes it possible to create rules for systematically transforming use cases into system sequence and interaction overview diagrams (Section 5).

Fig. 4.1 summarizes what we have discussed so far:

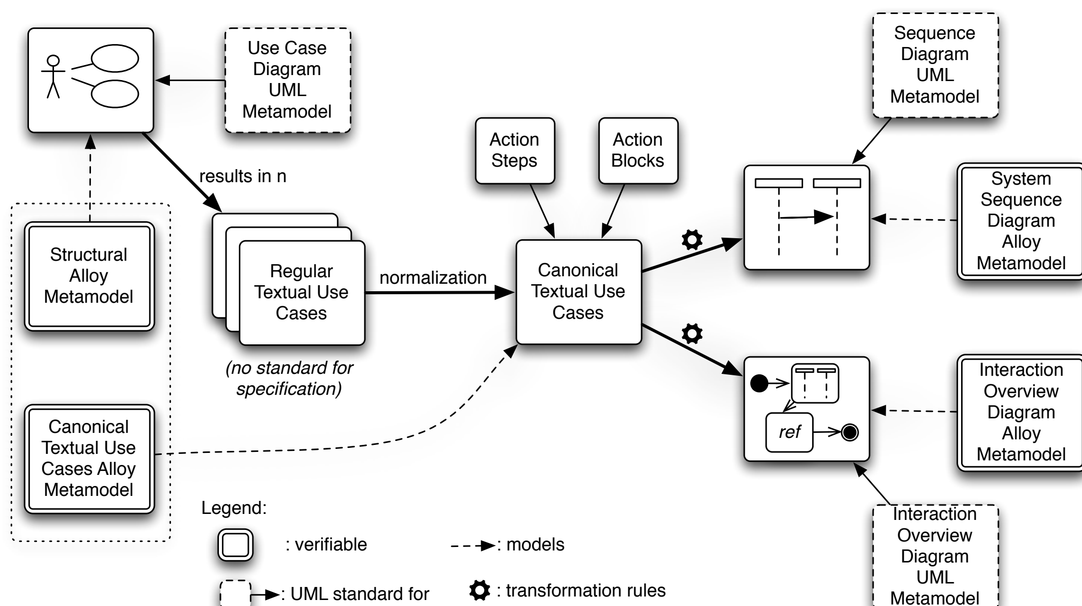


Figure 4.1: Graphical overview of our approach.

In conclusion, this chapter will show how the use case, system sequence, and interaction overview diagrams were modelled in Alloy. We take on one constructing element of each diagram at a time, first explaining how its structure was abstracted

in Alloy and then presenting the well-formedness rules of that element. This translation to Alloy is an essential step towards the final aim of being able to test the coherence of different UML models by taking advantage of the Alloy Analyzer.

4.1 Canonical Form Use Cases

After combining the structuring mechanisms introduced into the textual specification of use cases with the common knowledge about use case diagrams, we reached the meta-model depicted in Fig.4.2, where we highlight our contributions towards reaching a canonical form for use cases. It is important to note, however, that despite conferring regular textual use cases an additional structural layer, their semantic expressive power is retained by the resulting canonical use cases.

In our model, a use case diagram is basically composed of the system and the actors, whether they correspond to people or other software systems, that interact with it through use cases. Use cases invariably have a name, a main success scenario and a goal level, which may be *user goal* or *sub-function*. Furthermore, a use case may also have alternatives. If any of those alternatives are *external* that means they are contained in another use case which, in turn, means that the use case must also have extension points to account for those use cases that extend it. Flows are composed of action blocks and/or single steps like the *Goto* or *Include* steps. On the other hand, action blocks are composed only of *atomic* steps, which can be any action step, like an *Input*, *Output*, *SystemResponsibility*, *SystemCheck*, *InputValidation* or *UserDecision*. Based on which action steps actions blocks actually have, they can be of different types such as *Service*, *Validation*, *Query*, *Internal* or *SystemDependency*.

In the remainder of this section we shall elucidate what exactly is and how we reached what we designate as the canonical form of textual use cases; a prerequisite

to the transformation process proposed in this dissertation. Followed by looking in detail into each of these entities by referring to their Alloy code and eventual well-formedness rules.

4.1.1 Towards a Canonical Form for Use Cases

We now explain how to construct use cases so that they can be used in the transformation process proposed in this work. We start by defining the various kinds of steps a use case may be composed of. Some of these are action steps, which are identified afterwards, where we also explain how we refined the concept of assume action steps introduced in Helldahl and Ashraf [HA09]. Subsequently, we clarify the concept of action blocks and introduce and define generic kinds of action blocks that arise in practice. Finally, the user-goal and sub-function levels of abstraction for use cases are explained.

Flow Steps

Textual use cases are composed of steps of various kinds. They can describe an action by the system or user, mark the termination of an use case or denote the inclusion of another use case. In this section, we will give an overview of the different types a step may be of.

There are four kinds of steps that mark the end of a flow: *Goto*, *Resume*, *Success* and *Failure*. Goto steps are used only in alternative flows and denote the step of the flow which originated the alternative to execute next. Resume steps are similar to gotos. The difference is that the step pointed to by resume steps is always the one immediately after the step that originated the alternative. They are intended to be used in the main flows of inclusion and extension use cases.

Success steps denote, as the name implies, the successful termination of a use case. Failure steps, on the other hand, denote the unsuccessful termination of use

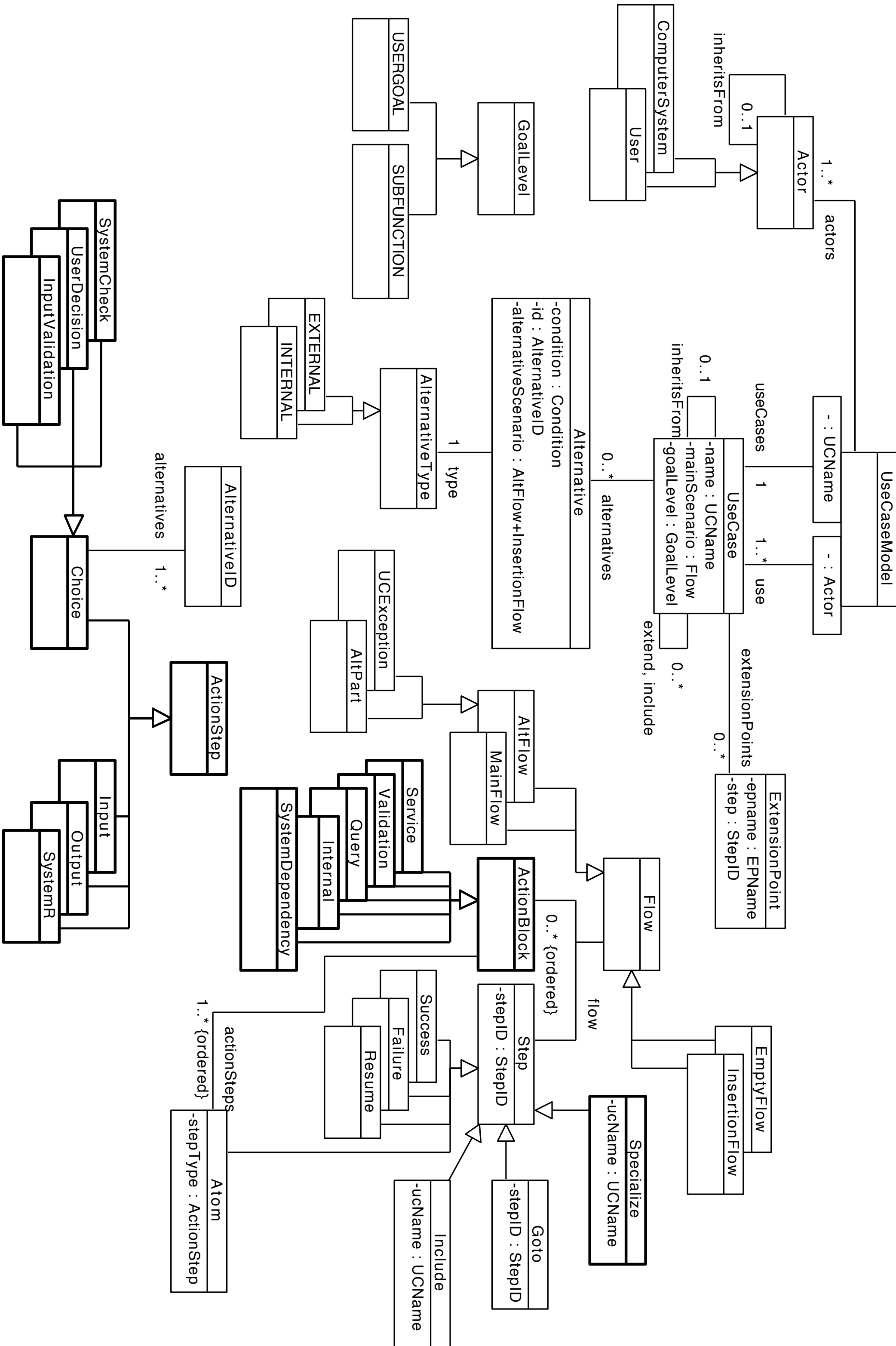


Figure 4.2: Use case meta-model.

cases and are used to model exceptions.

To signal that a use case includes or is extended by another at some point, include and extend steps are used, respectively. To specify that a use case, among a number of use cases with the same goal, will be executed, we use the specialize step.

Action Steps

We can distinguish between various kinds of atomic steps, the type of steps that describe the interaction itself. They can describe information or requests made by the user or results provided by the system, among others. Depending on the nature of atom steps, they are classified by associating them with *action steps*, which were introduced earlier.

As we also stated before, we did not find the existent kinds of action steps sufficient. After using the action steps defined in Helldahl and Ashraf [HA09] (i.e., input, assume, system responsibility and output) to classify the interaction steps in sample use cases, we concluded that assume steps did not fit well within the action block structuring mechanism when the 'assumption' was made based on a user decision. Steps 3 and 7 of the use case illustrated in Table 4.1 show the limitations of the assume step under these circumstances. They do not fit into existing action blocks, which are identified by the shaded lines. Also, they describe thoughts or decisions made by the user that are internal to him and have no impact on the system, thus not really being an action step.

However, as it can be seen in the same table, assume steps which are based on the state of the system fit nicely into action blocks (steps 5 and 9). Therefore, one can distinguish two kinds of assume steps, one for decisions made by the user and another for decisions made by the system. Additionally, we can further distinguish two different kinds of decisions made by the system, whether they are based on the

Table 4.1: Textual representation of the Order Product use case with Assume action steps.

	AS	Narrative
Main Success Scenario	INP	1. Customer specifies desired product category.
	OUT	2. System displays search results.
	ASS	3. Customer is satisfied with search results.
	INP	4. Customer selects a product.
	ASS	5. System validates availability of desired product.
	OUT	6. System displays purchase summary.
	ASS	7. Customer decides to purchase product.
	INP	8. Customer submits payment info.
	ASS	9. The payment was authorized.
	SR	10. System carries out payment.
	OUT	11. System provides a confirmation number.
		12. *Success.*
Alternative		**3a. Customer is not satisfied with search results:**
	INP	3a1. Customer repeats product search.
		3a2. *Goto 1.*
Exception		**5a. The desired product is unavailable:**
	OUT	5a1. System informs Customer the product is unavailable.
		5a2. *Failure.*
Exception		**7a. Customer decides to cancel the use case:**
	INP	7a1. Customer cancels the operation.
		7a2. *Failure.*
Alternative		**9a. The payment was not authorized:**
	OUT	9a1. System informs Customer the payment was declined.
		9a2. *Goto 8.*

validation of an input made by the user or a check of the system's internal state. Thus, we have a total of three different kinds of steps which originate alternative flows, the original idea behind assume steps.

We maintain the idea of having a special kind of step to clarify, in the main flow, where alternatives arise. However, since we find assume steps, as originally defined, somewhat limited, we have created a new kind of step to replace them, the *Choice* abstract action step. It is abstract in the sense that action steps are not directly of type Choice but rather of one of its specializations. These specializations are, thus, *Input Validation* action steps, used, as the name implies, to validate user inputs; *System Check* action steps, used to verify some property on the system's internal state; and *User Decision* action steps, used in situations where the user may chose one among multiple options, each of them resulting in a different user-system interaction.

The remaining action steps are straightforward. These are the *Input*, *Output* and *System Responsibility* action steps. Action steps where the user enters some values or requests some functionality from the system are classified as inputs and mark the beginning of a new action block. Output action steps are those where the system replies to the user, either by requesting some input or presenting some information. Often, though, the system has to perform some calculations or change the internal state before replying to the user, these kind of steps correspond to system responsibilities. To address the limitation of assume steps shown previously in Table 4.1, user decision action steps may also initiate action blocks.

So, in summary, there is now a total of six concrete action steps. Two distinct types that can be performed by actors, and four to be performed by the system:

- Action steps - Actor
  - Input (INP)
  - User Decision (UD)
- Action Steps - System
  - Input Validation (IVAL)
  - System Responsibility (SR)
  - System Check (SC)
  - Output (OUT)

Action Blocks

As mentioned before, action blocks group action steps. However, there are many different ways of composing action blocks. They may be started by input or user decision action steps and the remaining action steps may form any non-empty sequence of the other action steps, with the restriction that if an action block contains an input validation action step, it is the second one. In general, and as has already been stated, all action blocks have the form depicted in Fig. 4.3.

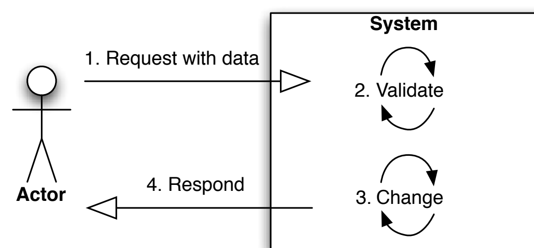


Figure 4.3: General structure of action blocks.

Nonetheless, considering the possible arrangements of action steps in action blocks, it is possible to differentiate five generic kinds of action blocks (Table 4.2). These are: *Validation*, *State Dependency*, *Query*, *Internal* and *Service* action

blocks. The only thing they all have in common is the first step which, as mentioned before, is always either an input or user decision.

Besides this first step, validation action blocks contain only one more step, an input validation. Due to the nature of its steps, action blocks of this type reflect situations where the system validates some input made by the user.

State dependency action blocks, on the other hand, may contain more than two steps. Specifically, this kind of action block contains some number of system checks and, possibly, input validations as well. The name state dependency reflects the necessity of performing some operation which relies on the state of the system, as indicated by the obligatory system check action steps.

Query action blocks express the intention of the user to get some information from the system or, alternatively, after the user has performed some input, the intention of the system to get some information from the user. Therefore, this kind of action blocks contains, mandatorily, at least one output action step and may or may not validate the user's input or verify the system's state for some condition.

Internal action blocks are characterized by the presence of system responsibility action steps and absence of any output. They typically model update, insertion or deletion operations, i.e., operations that change the internal state of the system. Naturally, the system may need to validate the user's input or check the current state before any change can be done. Therefore, internal action blocks may contain an input validation action step and an arbitrary number of system check action steps.

Finally, service action blocks potentially contain every kind of action step. Besides the action step that initiates the action block, system responsibilities and output are also mandatory. They are similar to internal action blocks in the sense that they also involve changes in the system's state. However, the intention is to

model an order given by the user to which an output is to be provided. Again, input validation and system check action step are optional.

Table 4.2: Structure of the different kinds of action blocks.

	Validation	**SD**	**Query**	**Internal**	**Service**
Input (or UD)	X	X	X	X	X
Input Validation	X	(X)[a]	(X)	(X)	(X)
System Check		X	(X)	(X)	(X)
SR				X	X
Output			X		X

[a] The parenthesis mean that the corresponding action step is optional in that action block.

Action blocks provide an easy and structured way to identify blocks of behavior that may be needed in the different use cases that compose an use case diagram. A critical analysis of the action blocks present in the textual description of a use case concerning their applicability in the other use cases of the domain provides a well structured method to discover possible bits of behavior that can be abstracted away in extending or included use cases, making the design more modular and robust.

Allowing user decision steps to initiate action blocks lets us rewrite the aforesaid use case in a more compact manner (Table 4.3). The assume action steps 3 and 7 are now removed and the following input steps (4 and 8) are changed to user decisions. This is because user decision action steps serve the double purpose of providing user input to the system and being a choice step at the same time, thus expressing that some alternative arises from it. The other two assume steps (5 and 9) are now input validations, which clarifies the intended semantics of the steps and is another advantage of the new action steps over the assume ones.

Furthermore, the use of action steps and action blocks is not limited or better

suited to single-column use cases, as the previous example might suggest. Any format to specify the textual representations of use cases works as long as we are able to identify the corresponding action steps and action blocks. In Fig. 4.3 we normalize the previous use case to conform to our canonical structure and present it in double-column format, a format commonly used in practice.

Goal Level

The structure of canonical use cases includes the definition of their level of abstraction. The UML specification [OMG11] states that use cases yield an observable result that is, typically, of value for the actor. Included and extending use cases are only a part of base use cases and, therefore, do not yield results of value to the user. Thus, base cases are on a higher level of abstraction than extending or included use cases.

Base use cases are considered to be on the *user-goal* abstraction level, while extending and included use cases are considered on the *sub-function* abstraction level. The abstraction level of base use cases is user-goal because they ultimately yield the result of value expected from successfully performing the use case. Additionally, user-goal use cases are always started by an action by the user, which means that an Input or UserDecision action step is always the first of this type of use cases.

On the other hand, sub-function level use cases do not necessarily start with a user action. Also, they do not yield a result of value to the user because they are a part of other use cases and not a whole in themselves. Hence, the name sub-function.

Table 4.3: New textual representation of the Order Product use case without Assume action steps and in double-column format.

	AB	AS	Actor	System
Main Success Scenario	Query	INP	1. Customer specifies desired product category.	
		OUT		2. System displays search results.
	Query	UD	3. Customer selects a product.	
		IVAL		4. System validates availability of desired product.
		OUT		5. System displays purchase summary.
	Service	UD	6. Customer submits payment info.	
		IVAL		7. The payment was authorized.
		SR		8. System carries out payment.
		OUT		9. System provides a confirmation number.
				10. *Success.*
Alternative			**3a. Customer is not satisfied with search results:**	
		INP	3a1. Customer repeats product search.	
				3a2. *Goto 1.*
Exception			**4a. The desired product is unavailable:**	
		OUT		4a1. System informs Customer the product is unavailable.
				4a2. *Failure.*
Exception			**6a. Customer decides to cancel the use case:**	
		INP	6a1. Customer cancels the operation.	
				6a2. *Failure.*
Alternative			**7a. The payment was not authorized:**	
		OUT		7a1. System informs Customer the payment was declined.
				7a2. *Goto 8.*

4.1.2 Use Case Diagram

Actor

The *Actor* signature models the entities which interact with the system, whether they are human users or other computer systems. The single field it contains denotes the actors it specializes, if any (Table 4.4). The `set` multiplicity keyword in the field declaration conveys that an actor can specialize any number of other actors.

Table 4.4: Entity *Actor*.

Description	Alloy
· **inheritsFrom**: identifies the actors this actor specializes, if any.	`sig Actor {` ` inheritsFrom: set Actor }`

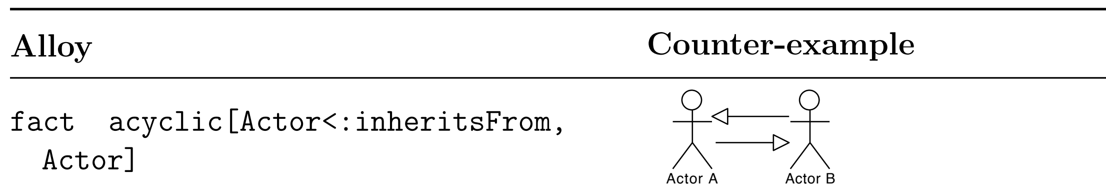
However, circular generalization between actors is prohibited since there is no point in two actors being the general and specialized versions of each other at the same time (Table 4.5). Besides, since each actor would be able to perform every use case the other actor could perform, we might as well just use one actor with access to all use cases. Still concerning the generalization between actors, we do not let the two types of actor specialize each other, i.e., human actors may not specialize machine actors and vice-versa:

```
fact { all u: User, s: ComputerSystem |
  u not in s.inheritsFrom and s not in u.inheritsFrom }
```

Use Cases

Use cases identify and describe each task a system is to able to perform. They are identified by their name and possess a main scenario (see Section 4.1.3 for details

Table 4.5: **Well-formedness rule #1: Generalization relation between actors is acyclic.**



Alloy	**Counter-example**
`fact acyclic[Actor<:inheritsFrom, Actor]`	Actor A, Actor B

on the *Flow* entity). They are also characterized by their goal level (presented in Section 4.1.1) which has implications on the way the use case may be used, as we will see later on. Naturally, a use case may also contain alternative flows (further analyzed in Section 4.1.3) to describe some processes that may happen but are less frequent than the main scenario. Besides these intrinsic properties, use cases may also be related to other use cases through inclusion, extension and generalization. As it can be seen on the Alloy code (Table 4.6), as far as generalization properties is concerned, each use case can at most have one parent, which assured by using the keyword v|lone|.

This means that each use case can specialize at most one other use case, thus preventing multiple inheritance between use cases. This decision was made considering that the specialization of use cases involves inheriting the goal of the parent use case and since a use case provides only *one* result of value to a user, inheriting from two or more use cases would mean having to provide multiple results.

Considering the includes and extends relations, however, it is accepted that a use case may include or extend multiple use cases.

Nevertheless, combining these relations may have ill effects. For instance, a use case including its parent use case would result in cyclic inclusions. Similarly, if a use case could extend its parent use case then, according to the substitutability principle, it would mean that it could extend itself which, while not resulting in an infinite cycle, would be better modeled using an internal alternative flow.

Table 4.6: Entity *UseCase*.

Description	Alloy

· **name**: use case's name.
· **goalLevel**: use case's goal level.
· **mainScenario**: use case's main flow.
· **alternatives**: use case's alternative flows.
· **extensionPoints**: use case's extension points.
· **inheritsFrom**: the use case this use case specializes, if any.
· **include**: the set of use cases this use case includes.
· **extend**: the set of use cases this use case extends.
	`sig UseCase {`
	`name: one UCName,`
	`goalLevel: one GoalLevel,`
	`mainScenario: one Flow,`
	`alternatives: set Alternative,`
	`extensionPoints:`
	`ExtensionPoint,`
	`inheritsFrom: lone UseCase,`
	`include: set UseCase,`
	`extend: set UseCase`
	`}`

Consequently, we introduced a well-formedness rule which states that the when a use case specializes another, then it cannot include or extend it (Table 4.7).

Table 4.7: **Well-formedness rule #2: A use case cannot include nor extend its parent use case.**

Alloy	Counter-example

`some uc: UseCase |`
`uc in inheritsFrom =>`
`uc not in include and uc not in extend`
	UseCase 1
	<<Include>>
	UseCase 2

However, abstract use cases can be extended. What this means semantically is that each of the specializing use cases is extended by the use case extending their parent (Table 4.8).

Comparably, if a concrete use case is extended by another, then it must define at least one extension point (see the Alloy modeling of extension points in Section

Table 4.8: **Well-formedness rule #3: Specializing use cases are extended by all use cases that extend their parent.**

<table>
<tr><th>Alloy</th><th>Counter-example</th></tr>
<tr><td>all uc: extend & abstractUseCases |
 some a: Alternative |
  a in uc.~@inheritsFrom.@alternatives
  and a.type in EXTERNAL and
  a.alternativeScenario in
   uc.~@inheritsFrom.@mainScenario</td><td>UseCase 1
<<Extend>>
UseCase 4
UseCase 2
extension points
UseCase 4
UseCase 3</td></tr>
</table>

4.1.3). In fact, a use case must define at least as many extension points as there are use cases extending it, since an extension use case may extend another use case at different places:

```
#extensionPoints >= #{ a: alternatives | a.type in EXTERNAL }
```

We have also identified some well-formedness rules concerning the consistency between use case diagrams and their textual specification. For every kind of relation represented in the diagram, there has to exist its counterpart in the textual specification. For instance, if a textual specification mentions the inclusion of an use case, that relation must be mirrored in the diagram[1]:

```
this.concreteIncludes in include
```

The opposite is also true, i.e., if a use case diagram relates two use cases with an inclusion relationship there has to be an inclusion step somewhere in the textual specification of the base use case referencing the included use case, as long as this one is concrete:

[1] *concreteIncludes* refers to the textual representation of inclusions while *include* refers to their diagrammatic counterpart

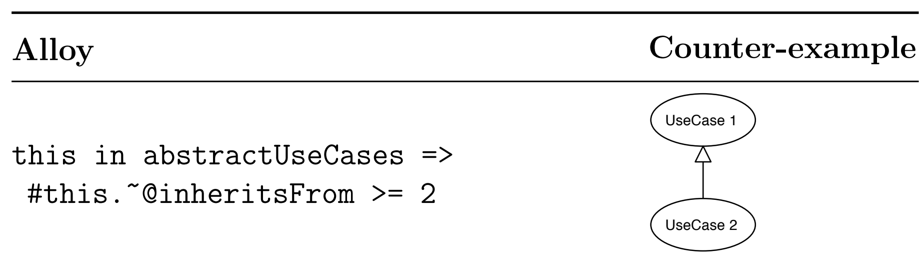
```
all uc: include - abstractUseCases | some i:Include |
 i in Int.(mainScenario.flow+alternatives.alternativeScenario.flow)
 and i.ucName in uc.@name
```

In case the included use case is abstract, then instead of an inclusion step there must be a specialization step:

```
all uc: include & abstractUseCases | some s:Specialize |
 s in Int.(mainScenario.flow+alternatives.alternativeScenario.flow)
 and s.ucName in uc.@name
```

Still concerning abstract use cases, we also imposed a restriction on the number of child use cases a use case may have. Since abstract use cases are not instantiable, as they do not even define a flow, it does not make sense for an abstract use case to have only one child use case. If this were the case, the specializing use case would replace its parent every time it was mentioned and one might just well work only with the former (Table 4.9).

Table 4.9: **Well-formedness rule #4: Abstract use cases are specialized by at least two use cases.**



Alloy	Counter-example
`this in abstractUseCases =>` ` #this.~@inheritsFrom >= 2`	UseCase 1 UseCase 2

Similarly to actors, circular generalization between use cases is disallowed (Table 4.10). Note that this restriction was not modeled as a signature fact, hence the use the `fact` syntax.

Likewise, use case diagrams with circular inclusions are not well-formed since they create infinite cycles (Table 4.11).

Table 4.10: **Well-formedness rule #5: Generalization relation between use cases is acyclic.**

Alloy	Counter-example
`fact acyclic[UseCase <: inheritsFrom, UseCase]`	

Table 4.11: **Well-formedness rule #6: Inclusion relation is acyclic.**

Alloy	Counter-example
`fact acyclic[include, UseCase]`	

And naturally, it is also not allowed to create inclusion cycles via use case textual steps as well:

```
fact { acyclic[concreteIncludes, UseCase] }
```

Well-formedness rules constraining the type of flow a use case may describe have been identified as well. Abstract use cases, for instance, do note have a textual representation of their flow (while concrete ones must have). In Alloy, this means that they implement the *EmptyFlow*:

```
this in abstractUseCases => mainScenario in EmptyFlow
  else mainScenario not in EmptyFlow
```

The goal level of a use case also influences the type of flow it may describe. User-goal level use cases, when successfully completed, always provide a result of value to the user. This means that the type of flow contained in user-goal level use

cases should be *MainFlow*, which is the same as saying the last step of a user-goal level use case should be the *Success* step:

```
goalLevel in USERGOAL => mainScenario in MainFlow + EmptyFlow
```

The reason the *mainScenario* of a user-goal level use case may also be an *EmptyFlow* is because we need to consider that actors may be associated with abstract use cases as well. In this situation, however, we also need to make sure the specializing use cases' goal level is likewise user-goal. In fact, a specializing use case should always inherit its parent's goal level:

```
some inheritsFrom => goalLevel in inheritsFrom.@goalLevel
```

Another restriction based on the type of flows a use case can have is imposed on included use cases. Included use cases are an integral part of the use case which includes them and do not represent any kind of alternative flow, they are a stepping stone in the path to the use case's success. Therefore, included use cases cannot describe scenarios leading to the failure of the including use case.

```
this in UseCase.concreteIncludes =>
  mainScenario not in UCException and all a: alternatives |
    a.alternativeScenario not in UCException
```

Furthermore, included use cases cannot be directly associated to actors since they do not provide any result of value, abstracting only a piece of behavior from another use case (Table 4.12). Hence, the goal level of included use cases is sub-function.

Use Case Model

The *UseCaseModel* entity identifies all use cases and actors used in the diagram and which actors perform which use cases (Table 4.13).

Table 4.12: **Well-formedness rule #7: Actors cannot be associated with included use cases.**

Alloy	Counter-example
`this in UseCase.@include =>` ` this not in UseCaseModel.use[Actor]`	System UseCase 1 <<Include>> UseCase 2 Actor

Table 4.13: Entity *UseCaseModel*.

Description	Alloy

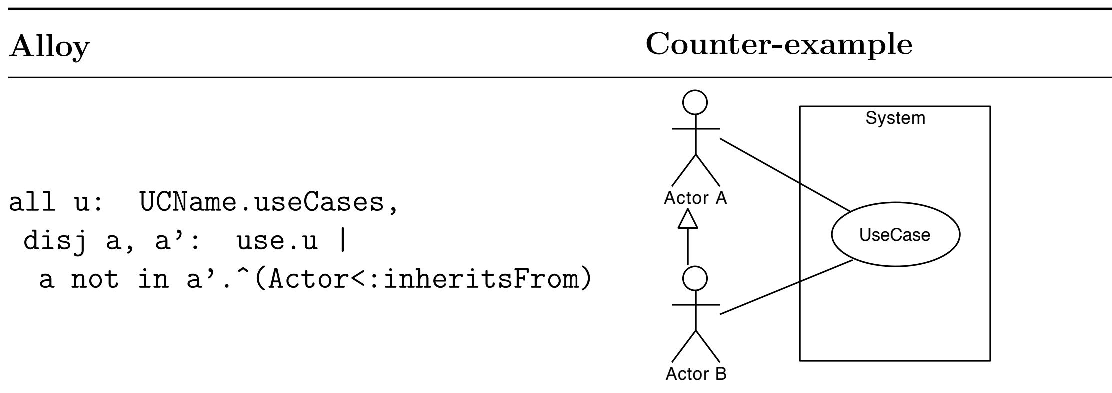
· **useCases**: maps use case names to use cases.
· **actors**: identifies the model's actors
· **use**: maps actors to the use cases they use
	`one sig UseCaseModel {`
	`  useCases: UCName -> UseCase,`
	`  actors: some Actor,`
	`  use: Actor set -> some UseCase`
	`}`

Using the keyword `some` we are already stating that a use case model invariably contains at least one actor and that each actor is directly related to a use case. This means that even if an actor could perform a use case by specializing another actor, and therefore being able to perform any use case the parent actor could, that actor also has be to able to perform some tasks that the parent actor could not, otherwise the child actor could be reduced to its parent.

Besides this rule, we identified other rules which were recorded as signature facts. The first of them prohibits a use case from being associated with two actors related with inheritance (Table 4.14). If an actor specializes another actor, then it automatically can perform any task the parent actor is able to perform, therefore, there is no need to associate an actor with a use case twice.

Likewise, it does not make sense to associate an actor with two use cases related with inheritance (Table 4.15). The generalization semantics of use cases state that

Table 4.14: **Well-formedness rule #8: A use case cannot be associated with two actors related with inheritance.**

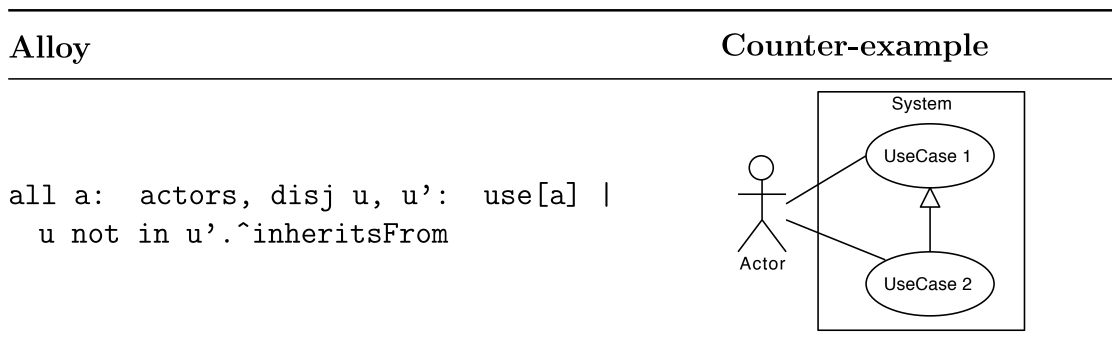


Alloy	Counter-example

`all u: UCName.useCases,`
` disj a, a': use.u |`
`  a not in a'.^(Actor<:inheritsFrom)`	

a child use case can substitute the parent use case anywhere the latter appears. Therefore, when an actor is able to perform a parent use case, it is automatically able to perform any of the child use cases as well.

Table 4.15: **Well-formedness rule #9: An actor cannot be associated with two use cases related with inheritance.**



Alloy	Counter-example
`all a: actors, disj u, u': use[a] |` `  u not in u'.^inheritsFrom`	

A third rule between use cases and actors states that an actor can only perform use cases whose goal level is *user-goal*:

```
all uc: use[Actor] | uc.goalLevel in USERGOAL
```

This is in accordance to the semantics of goal levels discussed in Cockburn [Coc00].

4.1.3 Textual Specification of Use Cases

Steps

As presented before, there are various kinds of steps. Steps can be *Include* steps, *Success* steps and *Failure* steps, among others. The nature of these steps is very different and the only property they all have in common, the *stepID*, is encapsulated in the *Step* abstract signature which they all specialize (Table 4.16).

Table 4.16: Entity *Step*.

Description	**Alloy**
· **stepID**: identifies the step.	`abstract sig Step {` `  stepID: one StepID }`

*Atom*s represent the atomic interaction steps used in the actor-system dialogue. These can be any of the action steps (Table 4.17), but not *Success* or *Failure* steps, for example.

Table 4.17: Entity *Atom*.

Description	**Alloy**

· **stepType**: declares what kind of action step the *Atom* instance corresponds to.	`sig Atom extends Step {`
	`  stepType:  one ActionStep`
	`}`

Another kind of step is the *Goto*. *Goto* steps are used in alternative flows to determine the step of the base flow to execute after the alternative flow finishes. Therefore, they must identify that step (Table 4.18).

Many years ago, programming languages were unstructured, they had no repetitive control structures like *while* or *repeat*, and the programmer had to use *if* and *goto* statements to implement iteration. Experience using those languages showed

Table 4.18: Entity *Goto*.

Description	Alloy
· **otherStepID**: identifies the step to execute next.	`sig Goto extends Step { otherStepID: one StepID }`

that unstructured use of those constructs quickly led to code that was hard to read and harder to maintain, what became known as *spaghetti code*. Likewise, the use of the *Goto* modeling construct must be well defined and restricted. Specifically, we formulated a well-formedness rule stating that *Goto* steps may only point to steps of the base flow, i.e, the flow which originated the the alternative containing the *Goto*:

```
let baseFlow =
  this.~(select13[flow]).~alternativeScenario.~alternatives.
    mainScenario.flow[Int] {
        otherStepID in (baseFlow.@stepID +
        (baseFlow.actionSteps[Int]).@stepID)
    }
```

In textual specifications, use cases may include other use cases via the *Include* step. Consequently, the *Include* entity identifies the use case to be included (Table 5.11).

Table 4.19: Entity *Include*.

Description	Alloy
· **ucName**: identifies the included use case.	`sig Include extends Step { ucName:  one UCName }`

However, there are restrictions on the type of use cases *Include* steps may

reference. Particularly, they cannot reference abstract use cases:

```
ucName not in abstractUseCases.name
```

To reference abstract use cases, we use the *Specialize* step (4.20). Its signature is similar to that of *Include* steps:

Table 4.20: Entity *Specialize*.

Description	Alloy
· **ucName**: identifies the specialized use case.	`sig Specialize extends Step {` `ucName: one UCName }`

Its signature fact, however, states the opposite of its *Include* equivalent, i.e., *Specialize* steps may only reference abstract use cases.

```
ucName in abstractUseCases.name
```

The last three types of steps do not possess any internal structure, acting mostly as labels. These are the *Success* step, which marks the successful termination of a use case; its opposite, the *Failure* step, which marks the unsuccessful termination of a use case; and the *Resume* step, which returns the execution to next step of the base flow:

```
lone sig Success extends Step {}
lone sig Failure extends Step {}
lone sig Resume extends Step {}
```

Action Steps

As was also stated before, there are multiple kinds of action steps, such as *Input*, *Output* and *SystemResponsibility*. These, however, do not have any kind of intrinsic

structure and are mainly used as labels. *Choice* action steps, on the other hand, encapsulate a property that identifies the alternatives which originate from *Choice* steps (Table 4.21).

Table 4.21: Entity *Choice*.

Description	**Alloy**
· **alternatives**: identifies the alternatives originating from a *Choice* step.	`abstract sig Choice extends ActionStep { alternatives: some AlternativeID }`

The action steps which specialize *Choice*, such as *SystemCheck*, *UserDecision* and *InputValidation*, however, do not define any more properties of their own and their signature is much like the three action steps mentioned before:

```
one sig Input extends ActionStep {}
one sig Output extends ActionStep {}
one sig SystemR extends ActionStep {}

sig UserDecision extends Choice {}
sig InputValidation extends Choice {}
sig SystemCheck extends Choice {}
```

The only thing to notice is that, while there is no need for the existence of more than a single instance of *Input*, *Output* and *SystemR* action steps, there are possibly multiple *UserDecision*, *InputValidation* and *SystemCheck* action steps in a model. This is due to the property that they inherit from *Choice* which possibly has different values for each instance. This also explains why the *Choice* steps are not defined as `one`, like their non-Choice counterparts.

Action Blocks

Comparable to *Choice* action steps, there are likewise different kinds of action blocks having a single property in common. That property consists precisely of the action steps that make up action blocks, which is defined as a sequence *Atom*s (Table 4.22).

Table 4.22: Entity *ActionBlock*.

Description	Alloy

· **actionSteps**: sequence of the atomic steps that compose an action block.	`abstract sig ActionBlock {`
	`  actionSteps: seq Atom`
	`}`

Nevertheless, the *ActionBlock* signature also defines some constraints common to all action blocks. Particularly, it states that: (1) the first step of action blocks must be either an *Input* step or a *UserDecision* step, (2) that they cannot appear in the remainder of the action blocks' body, and that (3) action blocks contain at least two steps.

```
first[actionSteps].stepType in Input + UserDecision        (1)
Input not in Int.(rest[actionSteps]).stepType              (2)
#actionSteps > 1                                           (3)
```

The different kinds of action blocks that exist are: *Query*, *Internal*, *Service*, *Validation* and *SystemDependency*. The differences between them consist in the types of action steps each contains. The composition of the action blocks as defined in their signatures is as follows.

The last step of *Query* action blocks is an *output* and intermediate steps can only be *Choices*:

```
sig Query extends ActionBlock {
} {
    last[actionSteps].stepType in Output
    SystemR not in Int.actionSteps.stepType
}
```

Internal action blocks are defined by not containing any *output* action step and having at least one *system responsibility* step:

```
sig Internal extends ActionBlock {
} {
    Output not in Int.actionSteps.stepType
    some s: SystemR | s in Int.actionSteps.stepType
}
```

Similar to *Internal* action blocks, *Service* action blocks also contain at least one *system responsibility* step. However, they allow *output*s:

```
sig Service extends ActionBlock {
} {
    some s: SystemR | s in Int.actionSteps.stepType
    some o: Output | o in Int.actionSteps.stepType
}
```

Validation action blocks are characterized by having a minimum of one *input validation* step, no *system responsibility* steps and no *Output* steps:

```
sig Validation extends ActionBlock {
} {
    some v: InputValidation | v in Int.actionSteps.stepType
```

```
        SystemR not in Int.actionSteps.stepType
        Output not in Int.actionSteps.stepType
    }
```

Finally, *system check* steps are always present in *SystemDependency* action blocks. *Output* and *system responsibility* action steps, however, are not:

```
    sig SystemDependency extends ActionBlock {} {
        some sc: SystemCheck | sc in Int.actionSteps.stepType
        SystemR not in Int.actionSteps.stepType
        Output not in Int.actionSteps.stepType
    }
```

Extension Points

Extension points consist of places in a flow where a use case can extend that flow. Therefore, extension points, which are identified by their name, just need to indicate the place at which a use case may be extended (Table 4.23).

Table 4.23: Entity *ExtensionPoint*.

Description	**Alloy**

· **epname**: extension point's name.
· **step**: identifies a step where a use can be extended.
	`sig ExtensionPoint {`
	`  epname: one EPName,`
	`  step: one StepID }`

Flows

There are many types of flows depending on their purpose, *MainFlow*s for describing main success scenarios, for example, and *ExceptionFlow*s for describing

exceptions are two of those. The entity *Flow* is an abstract entity that encapsulates the single property that is common to all kinds of flows, the flow itself (Table 4.24), which consists of a sequence of steps and action blocks.

Table 4.24: Entity *Flow*.

Description	Alloy
· **flow**: a sequence of steps and/or action blocks that make up the flow.	`abstract sig Flow {` `  flow:  seq Step + ActionBlock }`

The well-formedness rules related to flows constrain the kinds of steps that may go into each flow. The only restriction common to all is described as a signature fact in the *Flow* entity and states that steps *Goto*, *Success*, *Failure* and *Resume* cannot belong to the body of a textual specification, except for the last step:

```
all s: Goto + Success + Failure + Resume |
  s not in Int.(butlast[flow])
```

However, to know which step actually terminates which flow, we now look at the individual types of flow. *MainFlow* is the entity used in scenarios of user-goal level use cases, the use cases that provide a result of value to the user. Therefore, its last step is the *Success* step. Besides, *MainFlow*s cannot be composed of just the *Success* step and must contain at least one step more:

```
sig MainFlow extends Flow {} {
    last[flow] in Success
    #flow > 1
}
```

In contrast to the *MainFlow* there is the *UCException*, which marks a flow as unsuccessful. Consequently, the last step of exception flows is the *Failure* step:

```
sig UCException extends AltFlow {} { last[flow] in Failure }
```

Note that *UCException* extends *AltFlow*. This abstract entity simply states that its specializations have one or more steps in their flow:

```
abstract sig AltFlow extends Flow {} { #flow > 0 }
```

The other specialization of *AltFlow* is the *AltPart* entity. This entity is used describe alternative flows that eventually return to the base flow, which is done via the *Goto* step:

```
sig AltPart extends AltFlow {} { last[flow] in Goto }
```

The flow of abstract use cases, as previously mentioned, is defined by the *EmtpyFlow* entity. This entity simply states that its flow contains no steps:

```
one sig EmptyFlow extends Flow{} { #flow = 0 }
```

Inclusion use cases' flow always returns to the base use case at a specific position, the step immediately after the corresponding inclusion step. Put another way, after the execution of an inclusion use case, the base use case resumes its course. Thus, the last step of *InsertionFlow*s is the *Resume* step. This type of flow may also be used to describe the flow of extension use cases if they share the same semantics:

```
sig InsertionFlow extends Flow {} {
    last[flow] in Resume
    this in extendeConcreto.UseCase.mainScenario +
        UseCase.includesConcretos.mainScenario
    #flow > 1
}
```

Alternatives

Most use cases have alternative flows. These flows only execute in case some condition is true and their scenario can be of different types of flow depending on the nature of the alternative. Additionally, alternatives may be classified according to their *type*, *Internal* if they are contained in the use case, or *External* if another use case contains the flow (Table 4.25).

Table 4.25: Entity *Alternative*.

Description	Alloy

· **id**: identifies the alternative.
· **condition**: alternative's condition.
· **alternativeScenario**: alternative's scenario, which can be an *AltFlow* or an *InsertionFlow*.
· **type**: alternative's type.
	`sig Alternative {`
	`id: one AlternativeID,`
	`condition: one Condition,`
	`alternativeScenario: one AltFlow + InsertionFlow,`
	`type: one AlternativeType }`

Regarding this entity we identified the following well-formedness rule:

```
(some c : id.~alternatives | c in InputValidation) =>
    first[alternativeScenario.flow].stepType in
        SystemR + Output + SystemCheck
```

Which states that the first step of alternatives derived from *input validation* steps must be an *output*, *system responsibility* or *system check* step. An *input* step is not acceptable because, after an the validation of an input, the flow control is on the side of the system.

4.2 System Sequence Diagrams

Since we did not introduce any new concepts to sequence diagrams, its meta-model (Fig. 4.4) is smaller than its use case counterpart. In essence, every system

sequence diagram has an actor and a system, represented by two different lifelines, and messages exchanged between them. The messages contained in a system sequence diagram have a well defined order and may correspond, besides actual messages, to frames and references. In turn, *Frame* is an abstract entity which may correspond to the *Opt*, *Break*, *Loop* or *Alt* combined fragments. Each of these frames have at least one operand (the *Alt* frame has at least two) and one condition for each operand. An operand may have messages in the same way a system sequence diagram can, i.e., they can be actual messages, references or other frames. We now detail each of a system sequence diagram's entities and well-formedness rules.

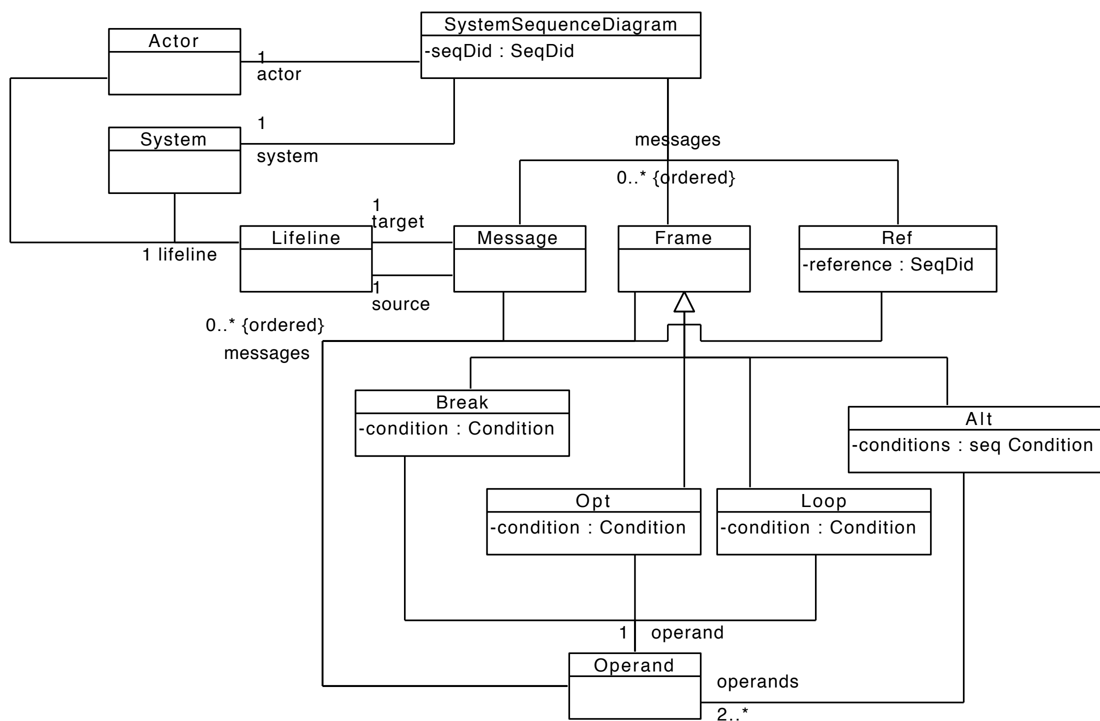


Figure 4.4: System sequence diagram meta-model.

4.2.1 Actor and System

Actor and *System* are the only two structures that may send and receive messages in system sequence diagrams. They are simply represented by a lifeline and there can be only one actor or system for each system sequence diagram, hence the use of the keyword `one` in their signatures (Table 4.26)

Table 4.26: Entities *Actor* and *System*.

Description	**Alloy**
· **lifeline**: the lifeline associated to the system.	`one sig System {` `  lifeline: one Lifeline }`
· **lifeline**: the lifeline associated to the actor.	`one sig Actor {` `  lifeline: one Lifeline }`

The *Lifeline* entity is a purely semantical one and has no internal structure:

```
sig Lifeline {}
```

4.2.2 Message

In system sequence diagrams, while the main dialogue consists of messages that are exchanged between the actor and system lifelines, the system is also able to send messages to itself. Either way, there is always a source and a target to each message (Table 4.27).

Table 4.27: Entity *Message*.

Description	**Alloy**

· **source**: the lifeline that sends the message.
· **target**: the lifeline that receives the message.
	`sig Message {`
	`  source: one Lifeline,`
	`  target: one Lifeline`
	`}`

4.2.3 Ref

Ref is UML's way of stating that a sequence diagram will now pass the execution control to another sequence diagram and the continue after the referenced sequence diagram is finished. Thus, the *Ref* entity only needs to record the identification of the diagram that is to be executed (Table 4.28).

Table 4.28: Entity *Ref*.

Description	Alloy
· **reference**: identifies the diagram which this frame references.	`sig Ref {` ` reference: one SeqDid }`

4.2.4 Frames

In this dissertation we consider four different kinds of frames: *Alt*, *Opt*, *Loop* and *Break*. All of which specialize the abstract *Frame* entity:

```
abstract sig Frame {}
```

Alt

The *Alt* combined fragment is intended to be used in situations where multiple flows may be executed, one of which must be selected. This selection is based on the evaluation of the guard condition of each operand, which is the structure that encloses the alternative flows. As may be observed in the Alloy code (Table 4.29), both the *operands* and the *conditions* fields are implemented using a sequence. The intention is to have them work similarly to parallel arrays, i.e., the condition at index n of the *conditions* sequence guards the operand at index n of the *operands* sequence, the condition at position m guards the operand at position m, etc.

Table 4.29: Entity *Alt*.

Description	Alloy

· **operands**: the sequence of operands contained in this combined fragment.
· **conditions**: the sequence of conditions corresponding to each operand.
	`sig Alt extends Frame {`
	`  operands: seq Operand,`
	`  conditions: seq Condition`
	`}`

Opt

The fields on the *Opt* signature are very similar to those of the *Alt* entity. The difference is that *Opt* frames only have one operand and, therefore, only one condition (Table 4.30).

Table 4.30: Entity *Opt*.

Description	Alloy

· **operand**: the operand contained in this fragment.
· **condition**: the condition corresponding to the operand.
	`sig Opt extends Frame {`
	`  operand: one Operand,`
	`  condition: one Condition`
	`}`

Even though their semantics differ substantially, *Break* and *Loop* combined fragments' definition is identical to Opt's, only the name of the signature changes.

Operand

Operands are the UML structures which capture the messages that go inside combined fragments. Therefore, the *Operand* entity contains only one property: *messages*. This property is defined as a sequence of actual messages, references to other system sequence diagrams and other frames (Table 4.31).

Table 4.31: Entity *Operand*.

Description	Alloy

· **messages**: the messages contained in the operand, which can actual messages, frames or references to other system sequence diagram.	`sig Operand {`
	`  messages: seq Message + Frame`
	`+ Ref }`

4.2.5 System Sequence Diagram

The *SystemSequenceDiagram* entity aggregates all the information pertaining to a system sequence diagram. Each diagram defines its own actor, system and the messages exchanged between them; this field is defined just like its homonym of the *Operand* entity, where *messages* can represent actual messages or message "containers" such as references to other diagrams or combined fragments (Table 4.32).

Table 4.32: Entity *SystemSequenceDiagram*.

Description	Alloy

· **seqDid**: identifies the diagram.
· **system**: represents the system.
· **alternatives**: represents the actor.
· **messages**: a sequence of messages between system and actor, which can be an actual message, a frame or a reference to another system sequence diagram.
	`sig SystemSequenceDiagram {`
	`  seqDid: one SeqDid,`
	`  system: one System,`
	`  actor: one Actor,`
	`  messages: seq Message + Frame`
	`+ Ref }`

Similarly to the inclusion of use cases, there is a constraint on system sequence diagrams which prevents cyclic references. This is done via the *references* function which returns all pairs of sequence diagrams where the first use case of the tuple references the second:

```
fact { acyclic[references, SystemSequenceDiagram] }
```

4.3 Interaction Overview Diagram

Considering that most of interaction overview diagrams' intricacies are related to sequence diagrams, much of it has already been explained. Therefore, the interaction overview diagram meta-model, after being stripped down of its sequence diagram elements, is rather simple (Fig. 4.5). Interaction overview diagrams were modeled using a semantics similar to that of linked lists. The *InitialNode* has a pointer to the next node in the flow, just like every node except the final nodes. The difference is that the initial node can point only to a decision node (*DecisionNode*), an embedded system sequence diagram (*IODSSD*) or a reference to a system sequence diagram (*IODREF*), and cannot point to any of the final nodes. On the other hand, these three kinds of nodes can point to any one of themselves plus the two kinds final nodes. However, the decision node is a special case since it can point to an arbitrary number of nodes, even though only one can be chosen during the execution of an interaction overview diagram. Which one to choose depends on the evaluation the guard condition associated to each possibility. Each of these elements will now be presented more thoroughly as well as some of the well-formedness rules imposed on interaction overview diagrams.

4.3.1 Interaction Overview Diagram

Due to the linked list-like semantics adopted for the modeling of interaction overview diagrams, the mapping the of interaction overview diagram concept to Alloy (Table 4.33) is straightforward and contains only a pointer to the diagram's initial node.

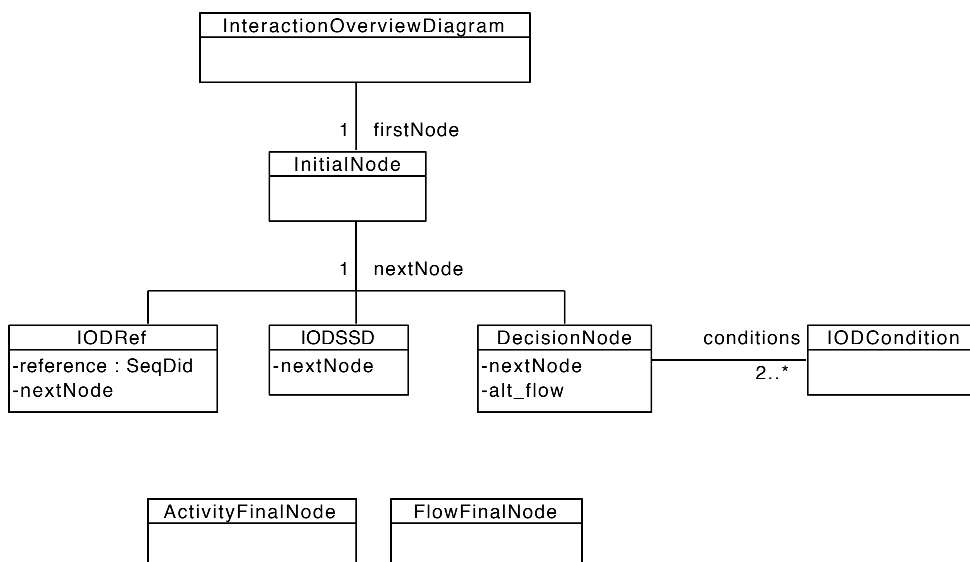


Figure 4.5: Interaction overview diagram meta-model.

Table 4.33: Entity *InteractionOverviewDiagram*.

Description	Alloy
· **firstNode**: node where the flow starts.	`one sig InteractionOverviewDiagram { firstNode: one InitialNode }`

4.3.2 Initial Node

The initial node marks the start of the execution of an interaction overview diagram. Naturally, there is only one initial node per diagram, which explains the use of the `one` keyword for defining the *InitialNode* entity (Table 4.34).

For the identification of the node following the initial node we use the *nextNode* field, which can point to either a reference to an external system sequence diagram, an embedded system sequence diagram or a decision node.

Table 4.34: Entity *InitialNode*.

Description	Alloy

· **nextNode**: points to the next node in the flow, which can be a reference to a system sequence diagram, an actual system sequence diagram or a decision node.	`one sig InitialNode {`
	`  nextNode:  one ( IODRef`
	`   + IODSSD + DecisionNode )`
	`}`

4.3.3 Decision Node

Decision nodes are the elements through which alternative flows are defined in interaction overview diagrams. When the execution of an interaction overview diagram reaches a decision node several conditions are evaluated. The condition that is evaluated as true determines the path to take, i.e., which node to execute next. For this reason, decision nodes contain two fields, *alt_flow* and *conditions*, both of which are modeled as sequences (Table 4.35). The types of the sequences are different though. The elements of the *alt_flow* sequence represent the nodes that may be executed next, which may be of the same types as the *nextNode* of initial nodes plus the two kinds of final nodes, *ActivityFinalNode* and *FlowFinalNode*. The *conditions* sequence, on the other hand, represents a sequence of conditions. These two sequences parallel the semantics of those in *Alt* frames, i.e., each condition in the *conditions* sequence guards the corresponding node in the *alt_flow* sequence.

Since the basic idea behind decision nodes is to model situations where one of several things may happen depending on the veracity of some conditions, decision nodes have at least two outgoing edges, otherwise there would be no decision to make (Table 4.36).

Furthermore, different outgoing edges must lead to different nodes or, again, there would be no real decision involved (Table 4.37). This restriction is imple-

Table 4.35: Entity *DecisionNode*.

Description	Alloy

· **alt_flow**: represents the flows which may be executed after this decision node.
· **conditions**: the conditions that guards each flow following this decision node.
	`sig DecisionNode {`
	`alt_flow: seq ( IODRef`
	`+ IODSSD + DecisionNode`
	`+ ActivityFinalNode`
	`+ FlowFinalNode )`
	`conditions: seq IODCondition }`

Table 4.36: **Well-formedness rule #10: Decision nodes have two or more outgoing edges.**

Alloy	Counter-example
`#alt_flow > 1`	[condition] ref

mented by equaling the length of the *alt_flow* sequence to the length of the set comprised of the nodes being pointed to. If the first were greater than the latter, it would mean that at least two different edges were pointing to the same node.

Table 4.37: **Well-formedness rule #11: Different outgoing edges of decision nodes must point to different nodes.**

Alloy	Counter-example
`#alt_flow = #Int.alt_flow`	[condition 2] [condition] ref

4.3.4 Ref

The *IODRef* entity is analogous to the *Ref* entity of system sequence diagrams. The only difference is that *IODRef* contains the *nextNode* field in addition the one that identifies the system sequence diagram to be executed (Table 4.38).

Table 4.38: Entity *IODRef*.

Description	Alloy

· **reference**: identifies the diagram which the frame references.
· **nextNode**: points to the next node in the flow, which may be another reference to a system sequence diagram, an actual system sequence diagram, a decision node, an activity final node or a final flow node.
	`sig IODRef {`
	`  reference: one SeqDid,`
	`  nextNode: one (IODRef`
	`   + IODSSD + DecisionNode`
	`   + ActivityFinalNode`
	`   + FlowFinalNode)`
	`}`

4.3.5 Sequence Diagram

The sequence diagrams embedded in interaction overview diagrams are very similar to those described before. In fact, the *IODSSD* entity inherits from *SystemSequenceDiagram* entity. Like most of the entities of interaction overview diagrams, though, it also defines a *nextNode* property which points to the node to be executed next (Table 4.39).

4.3.6 Activity and Flow Final Nodes

Activity and flow final nodes are the two nodes where interaction overview diagrams' flows end. Although their semantics differ, the activity final node indicates the successful termination of the flow whereas the flow final node indicates its failure, their implementation is identical. They do not contain any internal struc-

Table 4.39: Entity *IODSSD*.

Description	**Alloy**
· **nextNode**: the lifeline associated to the system points to the next node in the flow, which may be another reference to a system sequence diagram, an actual system sequence diagram, a decision node, an activity final node or a final flow node.	`sig IODSSD extends SystemSequenceDiagram { nextNode: one (IODRef + DecisionNode + IODSSD + ActivityFinalNode + FlowFinalNode) }`

ture or special constraints besides their multiplicity, which indicates there is one of each, at most, per interaction overview diagram:

```
lone sig ActivityFinalNode {}
sig FlowFinalNode {}
```

4.4 Conclusion

In this chapter we discussed how use cases, system sequence diagrams and interaction overview diagrams were modeled in Alloy. We also identified several well-formedness rules for each of the diagrams and explained why we thought they were appropriate. We made this translation so we can take advantage of the Alloy Analyzer.

Chapter 5

Transformation Rules

Transformation rules are an essential element in the systematic passage of a model to another. This process consists in isolating each construction of a source diagram and then apply the corresponding transformation rule to get the equivalent construct of the target diagram. The purpose is to obtain models consistent with each other to be able to propagate changes in a controlled manner every time the requirements change, from the system specification down to its implementation. In this context, this chapter addresses the transformation rules between use cases and system sequence diagrams in Section 5.1 and between use cases and interaction overview diagrams in Section 5.2. In Section 5.3, we provide a succinct example of the application of the transformation rules.

5.1 From Use Cases to System Sequence Diagrams

This section discusses the transformation rules between use cases and system sequence diagrams. Specifically, we will identify how to map action steps, alternative flows, exceptions, include relations, and extends relations to sequence diagrams.

5.1.1 Action Steps

Action steps are use cases' most basic constructions. It makes sense therefore to begin by examining the transformation rules of these so we can later elaborate more complex rules like the ones relating to alternative flows.

The representation of the actor-system dialogue is the central purpose of the textual specification of use cases. Thus, the elementary constructions of system sequence diagrams are the messages corresponding to the input and output action steps. Inputs correspond to requests made to the system or entering information required by it. Either way, inputs correspond to actions performed by the actor "on" the system. For this reason, in system sequence diagrams inputs are represented by a message originating in the actor and which has the system as the target (Table 5.1).

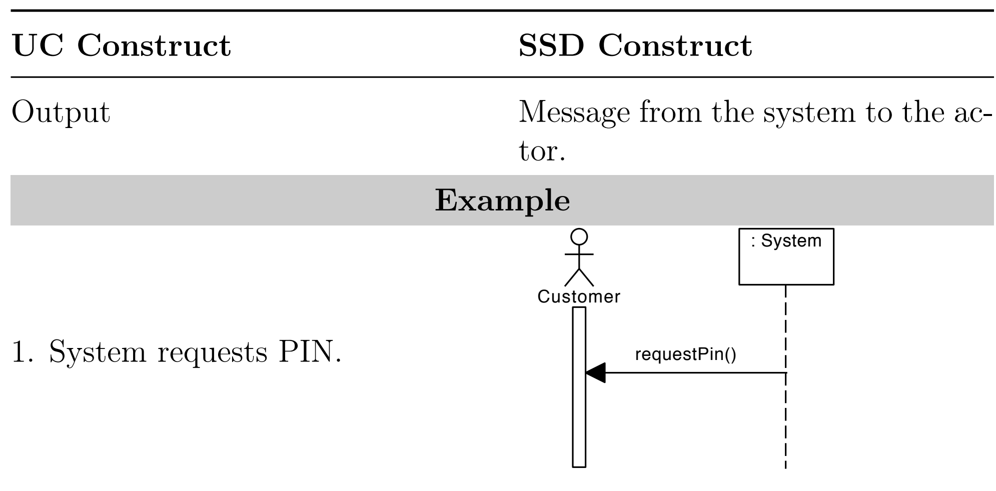
Table 5.1: **Transformation rule #1: Input steps to SSD.**

UC Construct	**SSD Construct**
Input	Message from the actor to the system, with or without parameters.
Example	
1. Customer enters PIN.	: System Customer enterPin(pin)

In contrast, output steps correspond to the request or presentation of information to the user, which, in any case, means that the system interacted with the user. Such steps, as presented in Table 5.2, are represented by messages sent by the system and received by the actor.

However, the actor-system interactions are not limited to these two types.

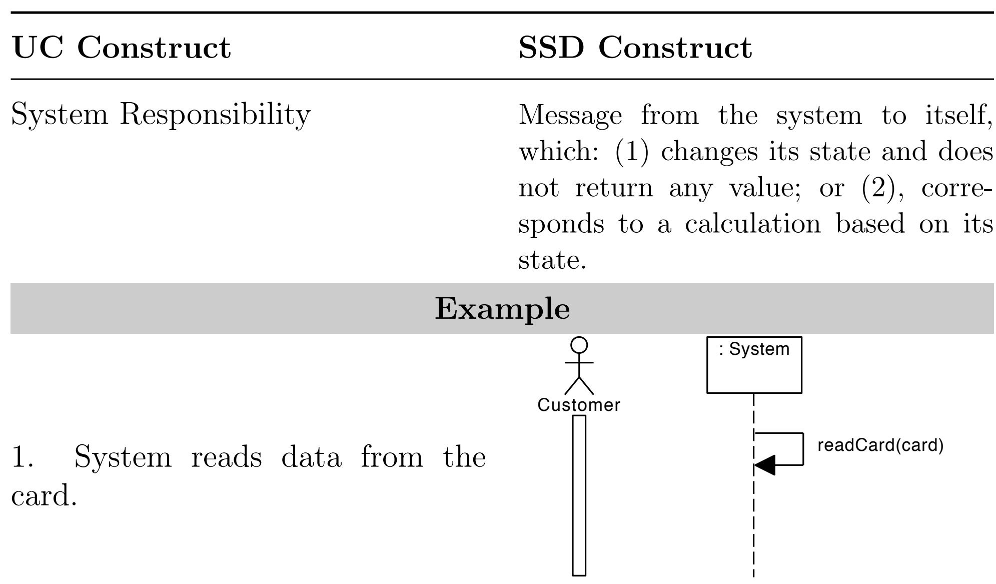
Table 5.2: **Transformation rule #2: Output steps to SSD.**



UC Construct	SSD Construct
Output	Message from the system to the actor.
Example	
1. System requests PIN.	

Sometimes the system needs to do some calculation or change its internal state before it can respond to a user request. These operations, corresponding to system responsibility action steps, are represented through system self-messages. Table 5.3 provides an example.

Table 5.3: **Transformation rule #3: SR steps to SSD.**



UC Construct	SSD Construct
System Responsibility	Message from the system to itself, which: (1) changes its state and does not return any value; or (2), corresponds to a calculation based on its state.
Example	
1. System reads data from the card.	

Just as sometimes it is necessary to perform some internal operation to modify

the system's state, in other occasions, it is necessary to refer to this state to be able to decide what action to take next. For example, if an ATM user attempts to withdraw a certain amount of money, the system must first check if it has that amount available before satisfying the user's request. Such checks correspond to the semantics of system check action steps. On the one hand, this kind of steps corresponds to a verification of the internal state of a system that is transparent to the user, which is why the are represented by a system self-message. On the other hand, they are always connected to an alternative flow. Reason why following the verification message, there is always a frame containing the alternative flow. The frame that should be used in each situation is indicated in Section 5.1.2.

This rule and summarized in Table 5.4.

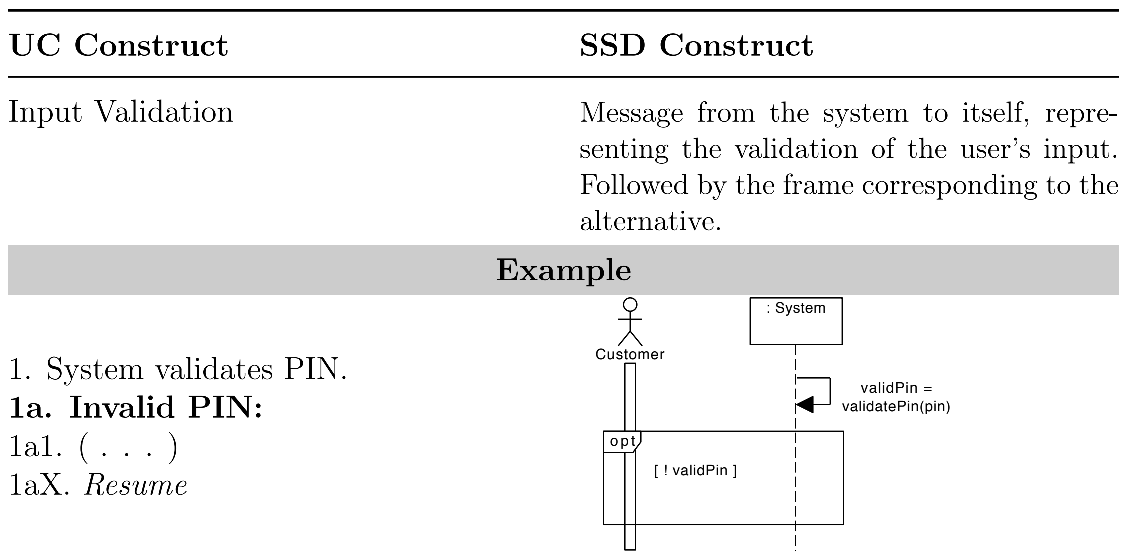
Table 5.4: **Transformation rule #4: SC steps to SSD.**

UC Construct	**SSD Construct**
System Check	Message from the System to itself, representing a state query. Followed by the frame corresponding to the alternative(s).
Example	
1. System checks whether it has enough money on hand. **1a. System does not have enough money on hand.** 1a1. (. . .) 1aX. *Resume*	

the input. Whatever the reason for the verification, the system behaves in different ways depending on its outcome.

As shown in Table 5.5, the mapping of these steps, which correspond to input validation action steps, is performed similarly to system check steps, since they are always associated with an alternative too.

Table 5.5: **Transformation rule #5: IVAL steps to SSD.**

UC Construct	**SSD Construct**
Input Validation	Message from the system to itself, representing the validation of the user's input. Followed by the frame corresponding to the alternative.
Example	
1. System validates PIN. **1a. Invalid PIN:** 1a1. (. . .) 1aX. *Resume*	

Not all user inputs are equal. Sometimes, the input is treated the same way every time. When a user identifies himself to a system, for example, the system will always check the validity of identity, whatever it may be. On other occasions, the action that the system performs is completely dependent on user input. Menus that display various options are a typical example, where, generally, each option is handled in a completely distinct manner.

This type of inputs, corresponding to user decision action steps, are mapped to system sequence diagrams through a message from the actor to the system of the form *userDecision(decision)*; where the *decision* parameter is a token that represents the choice made by the user. As the continuation of the execution of

the system sequence diagram depends on the choice of the actor, following this message there is usually one *Alt* frame, with as many operands as the options bestowed upon the user. The conditions that guard each alternative must be mutually exclusive and based on the value of the *decision* parameter.

Should any of the operands be empty, resulting, for example, from a branch that simply contains a *Resume* step, it is possible to omit that operand and use an *Opt* frame instead, in case there is only one remaining operand.

Specialize steps also correspond to user decisions. The nuance is that the user chooses an use case to perform. Thus, in each branch of the *Alt* interaction fragment there will be a *Ref* frame referencing one of the specializing use cases.

To clarify this transformation rule, we present an example in Table 5.6.

Table 5.6: **Transformation rule #6: UD steps to SSD.**

UC Construct	**SSD Construct**
User Decision	*userDecision(decision)* message from the actor to the system, followed by an *Alt* frame containing the flows of the different options. The guards should be mutually exclusive and based on the *decision* parameter.
Example	
1.System asks if the user wants to continue. 2. User indicates his choice. **2a. User wants to continue:** 2a1. (. . .) **2b. User does not want to continue:** 2b1. (. . .)	User : System continue?() userDecision(decision) alt [decision == yes] [decision == no]

5.1.2 Alternative Flows and Exceptions

The use cases do not always run smoothly executing only the main success scenario. When that happens, the execution path is diverted to alternative flows, later returning to the main flow; or to exception flows, which invariably leads to the unsuccessful termination of the use case.

Consequently, all alternatives end with the same type of step, the *Goto*; and all the exceptions end with the same step, the *Failure* step.

The transformation of exception flows, as shown in Table 5.7, is straightforward: the steps corresponding to the flow are captured in a *break* frame.

Table 5.7: **Transformation rule #7: exceptions to SSD.**

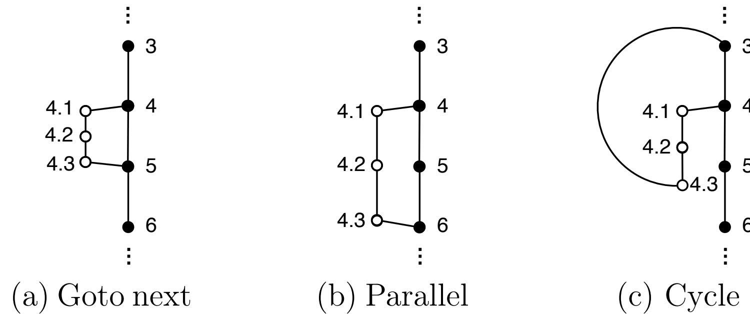
UC Construct	**SSD Construct**
Exception flow.	*Break* frame containing the exception flow's steps.
Example	
1a. Invalid card. 1a1. System ejects the card. 1a2. System notifies the user. 1a3. *Failure*	Customer : System break [! validCard] ejectCard() notifiyCardInvalid()

About the alternative flows, it is possible to distinguish three different types:

- Goto next or resume flows
- Parallel flows
- Cyclic flows

These flows are classified according to the nature of the *Goto* step that concludes them in relation to the step that leads to the alternative where the G*oto* is.

Specifically, if, for example, an action step of the type system check is the fourth step of a given use case and the alternative originated by it is concluded by a *Goto* step pointing to the step immediately after the system check, i.e., the fifth step, then we are dealing with a *goto next* or *resume* alternative flow. If, however, it points to a further step than the fifth, then the flow is said to be *parallel*. Finally, cyclic flows are characterized by a *Goto* step that points to a step prior to the one which causes the alternative, which in the case of the running example would be a step prior to the fourth. The three types of flows are pictured in Figure 5.1.



(a) Goto next (b) Parallel (c) Cycle

Figure 5.1: The three types of alternative flows.

The frame to be used to capture the alternative flow depends on the type of the flow. Alternative flows of the type represented in Figure 5.1a correspond to conditional insertions, i.e., simply correspond to the addition of behavior under certain conditions and do not involve the repetition or failure to perform any step of the main flow. Thus, after the message corresponding to the step that originates the alternative, an *Opt* frame that contains the alternative flow is inserted. This rule is synthesized in Table 5.8.

There are situations, however, in which the verification of a condition allows skipping some steps from the main flow, either because it no longer makes sense to employ these steps or because its application is no longer necessary. Whatever the reason, this type of situation originates the so-called parallel flows (Figure 5.1b).

Table 5.8: **Transformation rule #8: goto next/resume flows to SSD.**

UC Construct	**SSD Construct**
Goto step that points to the step immediately after the one which originated the alternative where the *Goto* is.	*Alt* or *Opt* frame containing the respective alternative flow.
Example	
1. Librarian indicates the return of a book. 2. System checks whether the delivery was made within the time limit. 3. System confirms the delivery. 4. *Success* **2a. Late delivery:** 2a1. System calculates the amount to be paid. 2a2. System provides fine value. 2a3. Librarian carries out payment. 2a4. *Goto 3* (or *Resume*)	Librarian : System returnBook(code) inTime = checkTimeLimit(code) opt [! inTime] fine = calculateFineValue() displayFine(fine) performPayment() confirmDelivery()

This type of flow involves performing one of two possible paths. If the alternative flow's guard condition is true, the alternative flow is executed instead of part of the main flow; if it is false, the main flow continues normally. For this reason, the *Alt* combined fragment is the proper fragment for the modeling of such flows. The fragment shall contain two operands: one of them containing the alternative flow and the other containing the part of the main flow skipped if the alternative is executed (Table 5.9).

Cyclic flows model situations where it may be needed to repeat some process. As can be seen in Figure 5.1c, part of the flow that may have to be repeated is part of the main success scenario. This means that, regardless the guard of the alternative is true or false, these steps are always executed at least once. This characteristic is reminiscent of the *do … while* structure several programming languages possess, where at the end of the execution of a block of code a condition

Table 5.9: **Transformation rule #9: parallel flows to SSD.**

UC Construct	SSD Construct
Goto step that points to a step posterior to the one which originated the alternative where the *Goto* is, skipping some steps from the main flow.	*Alt* frame with two operands. One of them will contain the alternative flow. The other will contain the skipped part.

Example

1. Customer wants to order the selected products.
2. System verifies that the customer is a new customer.
3. System requests shipping information.
4. Customer enters required information.
5. System requests credit card information.
6. Customer enters credit card data information.

2a. Customer is regular customer:
2a1. System displays shipping, pricing and billing information.
2a2. *Goto 5*

is evaluated and if it is true the block of code is executed again. However, the UML does not provide any combined fragment with this semantics. To work around this, we use the combined fragment *Loop*, but always keeping in mind that, to simulate the *do ... while* semantics, that the *Loop*'s entry condition is regarded as true the first time it is evaluated. In addition, the loop contains all messages from the main flow starting with the one that is referenced by the *Goto* and until the step of the alternative, as well as the alternative flow itself, which is enclosed in a fragment of its own. A typical example of this type of situations is shown in Table 5.10.

Table 5.10: **Transformation rule #10: cyclic alternatives to SSD.**

UC Construct	**SSD Construct**
Goto step that points to a step anterior to that which originated the alternative where the *Goto* is.	*Loop* frame that encompasses everything from the step referenced by the *Goto* step up to the step of the alternative, inclusive; the alternative being included in an *Alt* or *Opt* frame.
Example	
1. Secret agent enters the code. 2. System validates the code. 3. System launches warhead. 4. *Success* **2a. Invalid Code:** 2a1. System requests the code again. 2a2. *Goto 1*	

5.1.3 Include and Extends Relations

The passage of inclusion steps to sequence diagrams is trivial as this type of diagram has a construct with identical semantics: the *Ref* frame . Thus, whenever a use case references another use case via an inclusion step, such corresponds, in the sequence diagram, to the *Ref* frame indicating the included use case (Table 5.11).

References to use cases performed via the extension steps are treated analogously. One should just be conscious that, just as extension steps can be present only in alternative flows, also *Ref* frames corresponding to these steps must be contained within an *Alt* or *Opt* frame, depending on the type of alternative. Table 5.12 uses a similar example to that of Table 5.8 but where the alternative flow was abstracted in another use case.

Table 5.11: **Transformation rule #11: inclusion steps to SSD.**

UC Construct	SSD Construct
Use case inclusion step.	*Ref* frame.
Example	
1. *Include:* Withdraw Money	Customer : System ref Withdraw Money

Table 5.12: **Transformation rule #12: extension steps to SSD.**

UC Construct	SSD Construct
Use case extension step.	*Ref* frame inside an *Alt* or *Opt* frame.
Example	
1. Librarian indicates the return of a book. 2. System checks whether the delivery was made within the time limit. 3. System confirms the delivery. 4. *Success* **2a. Late delivery:** 2a1. *Extended by:* Handle Fine	Librarian : System returnBook(code) inTime = checkTimeLimit(code) opt [! inTime] ref Handle Fine confirmDelivery()

5.2 From Use Cases to Interaction Overview Diagrams

This section will identify the transformation rules that allow the systematic passage of use cases to interaction overview diagrams. Instead of action steps, this transformation focuses on action blocks. Thus, we begin by demonstrating how

to transform action blocks into the constructs of interaction overview diagrams, subsequently addressing the transformation of alternative flows and exceptions, and the include and extends relations among use cases.

5.2.1 Action Blocks

Owed to the characteristics of interaction overview diagrams, the transformation of use cases into these is performed at a level of abstraction a bit higher than the transformation into sequence diagrams. This means that instead of performing the transformation action step by action step it will be made on an action block by action block basis.

For transformation purposes, we can distinguish three kinds of action blocks:

- Simple action blocks
- Action blocks with alternatives
- Action blocks initiated by a user decision

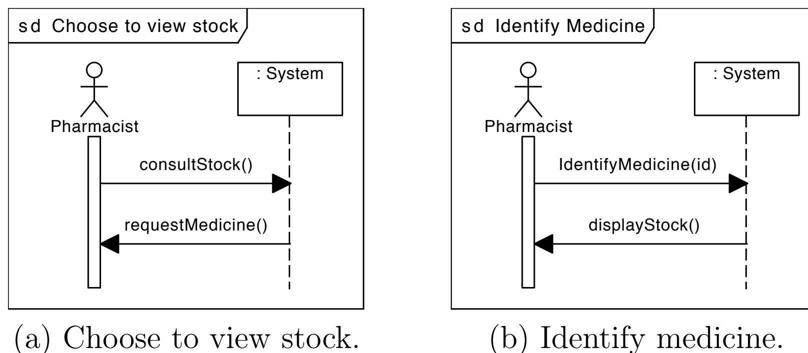
Simple action blocks consist of linear flows; i.e., none of its steps can originate alternative flows. Thus, we can abstract each of these action blocks in external sequence diagrams, being sure that we are not concealing paths relevant to the supervision of the functioning of the use case. The example in Table 5.13 contains two simple action blocks. One composed of steps 1 and 2 (input and output, respectively) and another composed of steps 3 and 4 (also input and output, respectively).

As we can see, each action block was transformed into a reference to the external sequence diagram that details the steps of the action block. To complete the example, we present these sequence diagrams in Figure 5.2.

In cases where, owed to system check or input validation action steps, an action block leads to alternative flows, this action block will have to be divided as many

Table 5.13: **Transformation rule #13: simple action blocks to IOD.**

UC Construct	**IOD Construct**
Simple action block, i.e., with no Choice steps.	*Ref* frame that references an SSD containing the messages corresponding to the steps of the action block.
Example	
1. Pharmacist indicates that he wants to view the stock of a medicine. 2. System asks which is the desired medicine. 3. Pharmacist enters the medicine's identification. 4. System shows the existing stock for the chosen medicine. 5. *Success*	ref Choose to view stock ref Identify Medicine



(a) Choose to view stock. (b) Identify medicine.

Figure 5.2: External system sequence diagrams.

times as the number of input validation plus system check steps that the action block contains. Except for the last frame, the last message of each of the frames resulting from the division, be it either a reference to an external sequence diagram or an embedded sequence diagram, will correspond to the proper Choice action step. After each of these frames, there comes a decision node.

In the example of Table 5.14, we can observe a use case with an action block that consists of three action steps, where the second one corresponds to the validation of

the user's input and, therefore, originates to an alternative flow. Thus, this action block is divided into two frames. The first, containing the first two messages, and the second, containing the third. These two frames are connected via a decision node, which is necessary because at this point in the execution, resulting from the existence of the system check step, there are two possible ways to go and the path to choose depends on the evaluation of the guards.

Table 5.14: **Transformation rule #14: action blocks with alternatives to IOD.**

UC Construct	**IOD Construct**
Action block with alternative flows.	Originates $1{+}n$ frames, where n is the number of IVAL + SC steps present in the action block. The last message of the n first frames is the message corresponding the appropriate *Choice* step. After each one of the first n frames, there is a decision node.
Example	
1. Librarian indicates the return of a book. 2. System checks whether the delivery was made within the time limit. 3. System confirms the delivery. 4. *Success* **2a. Late delivery:** 2a1. System calculates the amount to be paid. 2a2. System provides fine value. 2a3. Librarian carries out payment. 2a4. *Goto 3* (or *Resume*)	

The transformation of action blocks started by a user decision is similar to the previous case. The message corresponding to a user decision is always the last of the frame where it is contained and after that there is always a decision node. The peculiarity of this transformation as compared with the previous one lies in the message concerning the user decision step, whose parameter is used in the

guards of the decision node. Table 5.15 provides an example and summarizes this transformation rule.

Table 5.15: **Transformation rule #15: action blocks initiated by a UD to IOD.**

UC Construct	**IOD Construct**
Action block started by a UD	Decision node after the *userDecision(decision)* message, forking on the possible user choices.
Example	
1. System asks if the user wants a receipt. 2. User indicates his option. *(decision = Yes)* 3. System prints receipt. 4. *Resume* **2a. User does not want a receipt: *(decision = No)*** 2a1. *Goto 4*	

5.2.2 Alternative Flows and Exceptions

Unlike the transformation to sequence diagrams, the characteristics of interaction overview diagrams permit that the way of transforming alternative flows be always the same. Specifically, regardless of where the *Goto* step points to, in interaction overview diagrams, it is always represented by an arrow originated from the frame where the *Goto*'s preceding message is and whose destination is the frame where the message which corresponds to the action step the *Goto* points to is.

As the example of Table 5.16 shows, it is not always possible, just by reading the

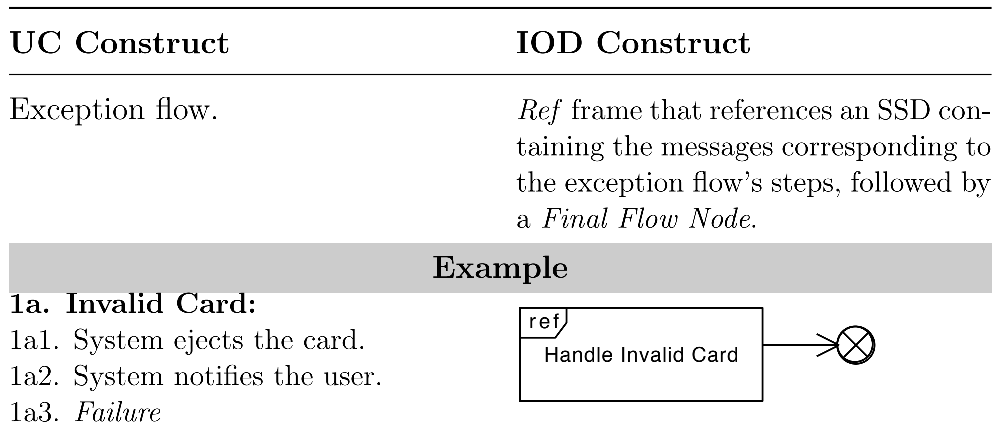
interaction overview diagram, to know the exact step, within a sequence diagram, to which the *Goto* points. This would be possible if one decomposed the frame that contains the step pointed to in several frames in order to, via the arrow, identify the step pointed to unambiguously. However, along with the other rules that already perform the decomposition of action blocks under certain conditions, this could lead to an explosion of tiny frames, which would make the interaction overview diagram overcrowded and the reading more difficult. For this reason, a commitment was made that introduces some ambiguity in the interaction overview diagram in exchange for keeping their construction and reading simple. In any case, this ambiguity can be undone by looking at the transformed use case.

Table 5.16: **Transformation rule #16: *Goto* steps to IOD.**

UC Construct	**IOD Construct**
Goto step.	Arrow with origin in the frame containing the message corresponding to the action step which precedes the *Goto* step and destination in the frame containing the message corresponding to the action step for which the *Goto* points to.
Example	
1. Secret agent enters the code. 2. System validates the code. 3. System launches warhead. 4. *Success* **2a. Invalid Code:** 2a1. System requests the code again. 2a2. *Goto 1*	

The transformation of exceptions into interaction overview diagrams, like into system sequence diagrams, is relatively simple. The messages corresponding to the steps of the exception flow are encompassed in an external sequence diagram and the interaction overview diagram merely references this diagram. However, to mark the failure of the use case, we adapt the semantics of the *final flow node*. That is, in the context of this work, the *final flow node*, which is characterized by a circle with an 'x' in the center, symbolizes the end of a flow that leads to the failure of the use case.

Table 5.17: **Transformation rule #17: exceptions to IOD.**



UC Construct	**IOD Construct**
Exception flow.	*Ref* frame that references an SSD containing the messages corresponding to the exception flow's steps, followed by a *Final Flow Node*.
Example	
1a. Invalid Card: 1a1. System ejects the card. 1a2. System notifies the user. 1a3. *Failure*	ref Handle Invalid Card

5.2.3 Include and Extends Relations

The include and extends steps, also similarly to the transformation into sequence diagrams, are treated identically. Each of them corresponds to a frame that references the proper use case. The difference between the two is the fact that before all *Ref* frames relating to extensions, there will be a decision node. Tables 5.18 and 5.19 summarize the rules of inclusion and extension, respectively.

Table 5.18: **Transformation rule #18: inclusion steps to IOD.**

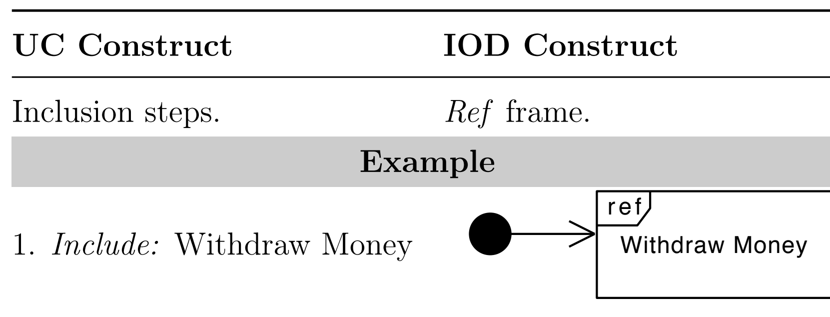
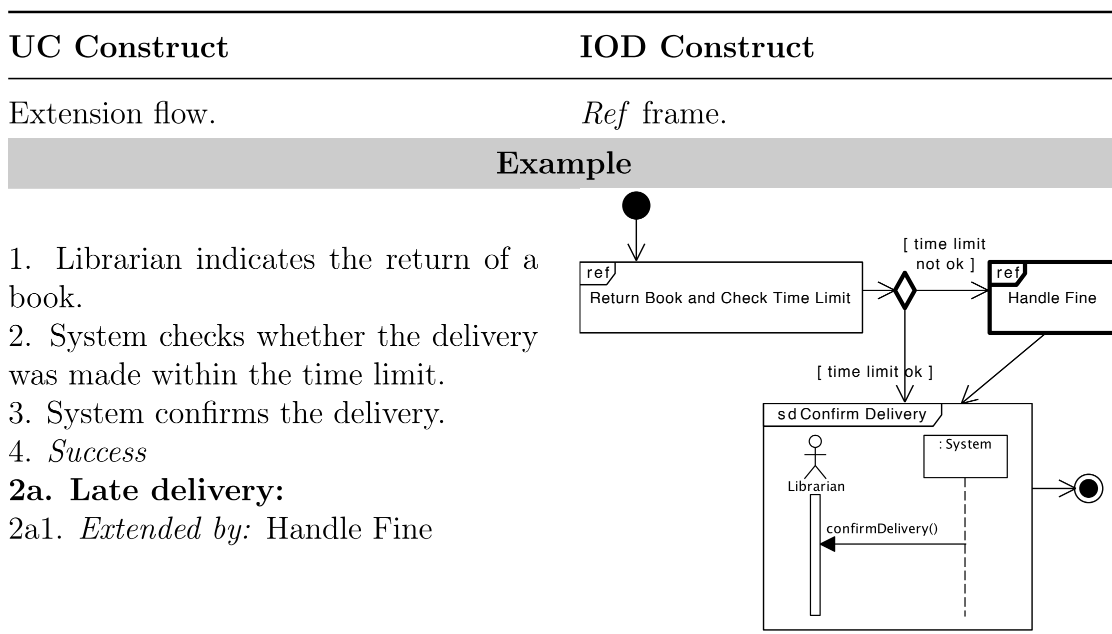
UC Construct	IOD Construct
Inclusion steps.	*Ref* frame.
Example	
1. *Include:* Withdraw Money	

Table 5.19: **Transformation rule #19: extension steps to IOD.**

UC Construct	IOD Construct
Extension flow.	*Ref* frame.
Example	
1. Librarian indicates the return of a book. 2. System checks whether the delivery was made within the time limit. 3. System confirms the delivery. 4. *Success* **2a. Late delivery:** 2a1. *Extended by:* Handle Fine	

5.3 Application of the Transformation Rules

The use case to system sequence diagram transformation rules can be applied in two distinct ways. One, as the definition of the rules might suggest, is action step-oriented; the other is flow-oriented.

The former is characterized by taking each step of the use case in a sequence, starting with the first, and applying the corresponding transformation rule. This

is the more direct way but may result in redrawing part of the diagram in case a loop is found since one has to include previously drawn messages in the *Loop* frame. We explain this method in-depth later on in the case study (Section 6).

The latter involves the construction of auxiliary intermediate diagrams. Concretely, a flow diagram and a frame schema. This approach is more visual and allows an early identification of loops as well as the other types of alternative flows and exceptions. The idea is to build the structure of the sequence diagram first and only subsequently include the messages.

Both approaches lead to the same end result; so, either one may be used. However, depending on the context, it may make sense to use one over the other. In complex scenarios, for example, the flow-oriented approach may prove to be more useful due to its visual nature, which aids in understanding the execution of the use case, and due to the additional documentation, which may be a helpful future reference. In contrast, for simpler use cases, the overhead introduced by building intermediate diagrams is probably unnecessary and the more direct action step-oriented approach is more appropriate.

The transformation from use cases to interaction overview diagrams is performed, as stated before, on an action block basis. Similarly to the aforementioned action step-oriented process, the transformation into interaction overview diagrams starts with the action block that initiates the use case, if there is one (extension use cases, for example, might not start with an action block); otherwise, the process is started with the first action step. The corresponding rule is applied and then the process is repeated for each of the successive action blocks (or action steps) until the end of the use case is reached.

To succinctly illustrate the transformation procedure into both system sequence and interaction overview diagrams we will consider a use case describing the service payment option of an ATM (Table 5.20). We use the colour red to highlight excep-

tions and the colour orange to highlight alternative flows. Since in this instance we shall use the flow-oriented approach for the transformation into a sequence diagram, these colours will then be used in the subsequent flow diagram and frame schema to easily identify all alternative flows and exceptions. We also colour the action blocks to better see the correspondence in the resulting interaction overview diagram.

Starting with the transformation into a sequence diagram and based on the textual specification of the use case, we build a flow diagram (Fig. 5.3). A flow diagram abstracts the details of the action steps and instead only refers to their relative order, i.e., the order in which they are executed. This is ideal for an early identification of all alternatives and exceptions and, since we also label include, extend, and specialize steps, to recognize all use cases the one the flow diagram represents interacts with.

The next step is to build the frame schema (Fig. 5.4). The frame schema depicts the structure of the final sequence diagram (i.e., the frames that constitute it) and its construction is based on the previous artefacts, i.e., the canonical textual specification of the use case and the flow diagram. Since the flow diagram intuitively illustrates loops, exceptions and other alternative flows, the choice of frame to represent each one is made easier.

Once the frame schema is completed, all there is left is to incorporate the messages corresponding to each action step (Fig. 5.5). The definition of the transformation rules can be consulted to find out the direction of the arrows depicting each action step, but note that the part of the rules that states what and how frames should be used is not to be applied since the previous steps of the flow-oriented approach to constructing sequence diagrams have taken care of that.

Concerning the transformation into interaction overview diagram, the result is always constructed directly from the use case by applying the transformation

Table 5.20: Textual specification of the Service Payment use case.

	AB	AS	Actor	System
Main Success Scenario	Query	INP	1. Customer selects the service payment option.	
		OUT		2. System requests payment data.
	Query	INP	3. Customer enters payment data.	
		IVAL		4. System validates the data.
		OUT		5. System displays the data and requests confirmation.
	Input Val.	UD	6. Customer enters his choice.	
		IVAL		7. System verifies if the amount is less than or equal to the account balance.
				8. *Include*: Register Transaction.
		OUT		9. System asks if customer wants a receipt.
		UD	10. Customer enters his choice.	
				11. *Success*.
Alternative			**4a. Invalid Data:**	
		OUT		4a1. System displays a data error message.
				4a2. *Goto 2*.
Exception			**6a. Customer cancels the operation** (decision = *Cancel*)**:**	
		OUT		6a1. System displays error message.
				6a2. *Failure*.
Exception			**7a. Insufficient balance:**	
		OUT		7a1. System displays error message.
				7a2. *Failure*.
Alternative			**10a. Customer wants a receipt** (decision = *Yes*)**:**	
				10a1. *Extended by*: Print Receipt.
				10a2. *Goto 11*.

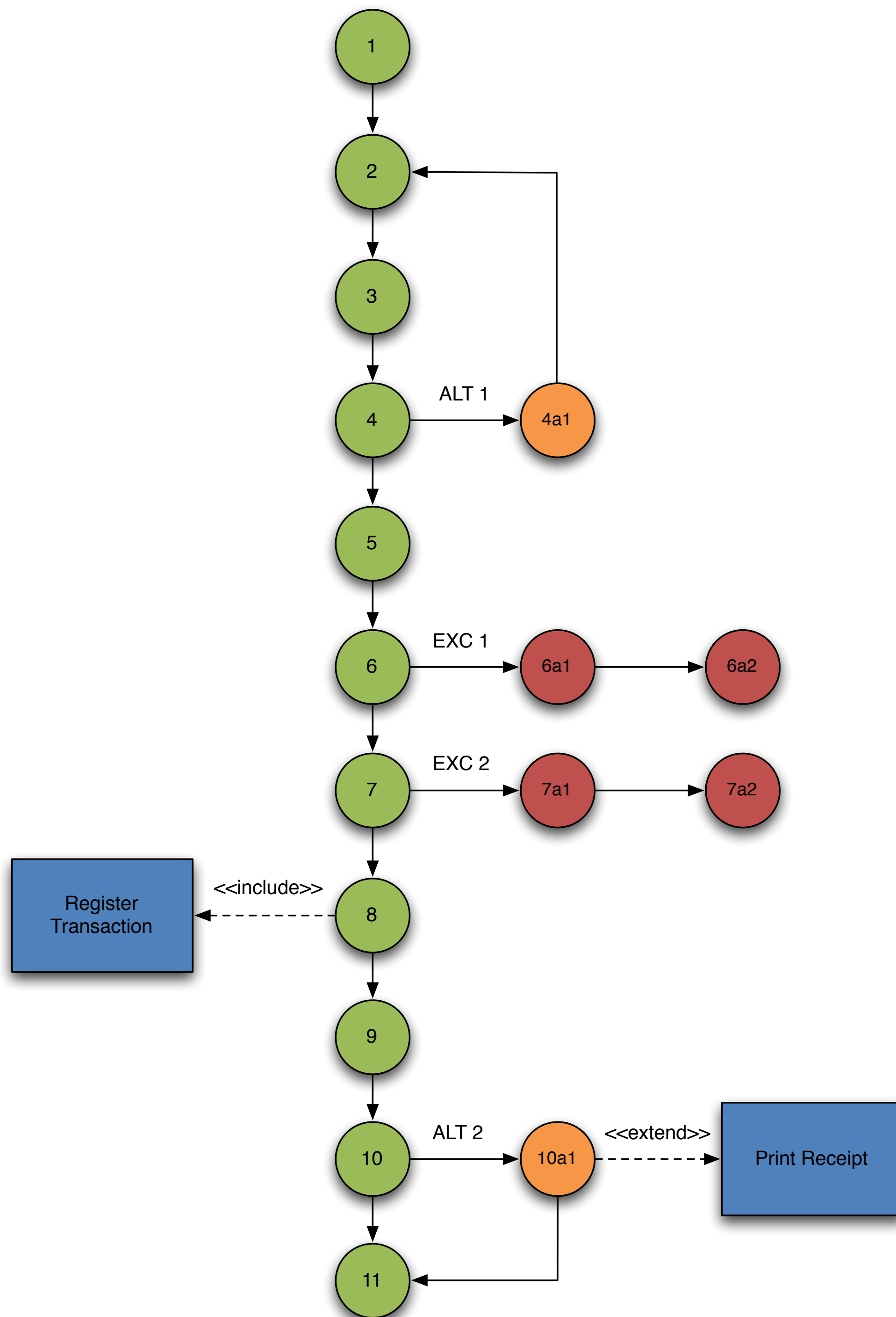


Figure 5.3: Flow diagram of the Service Payment use case.

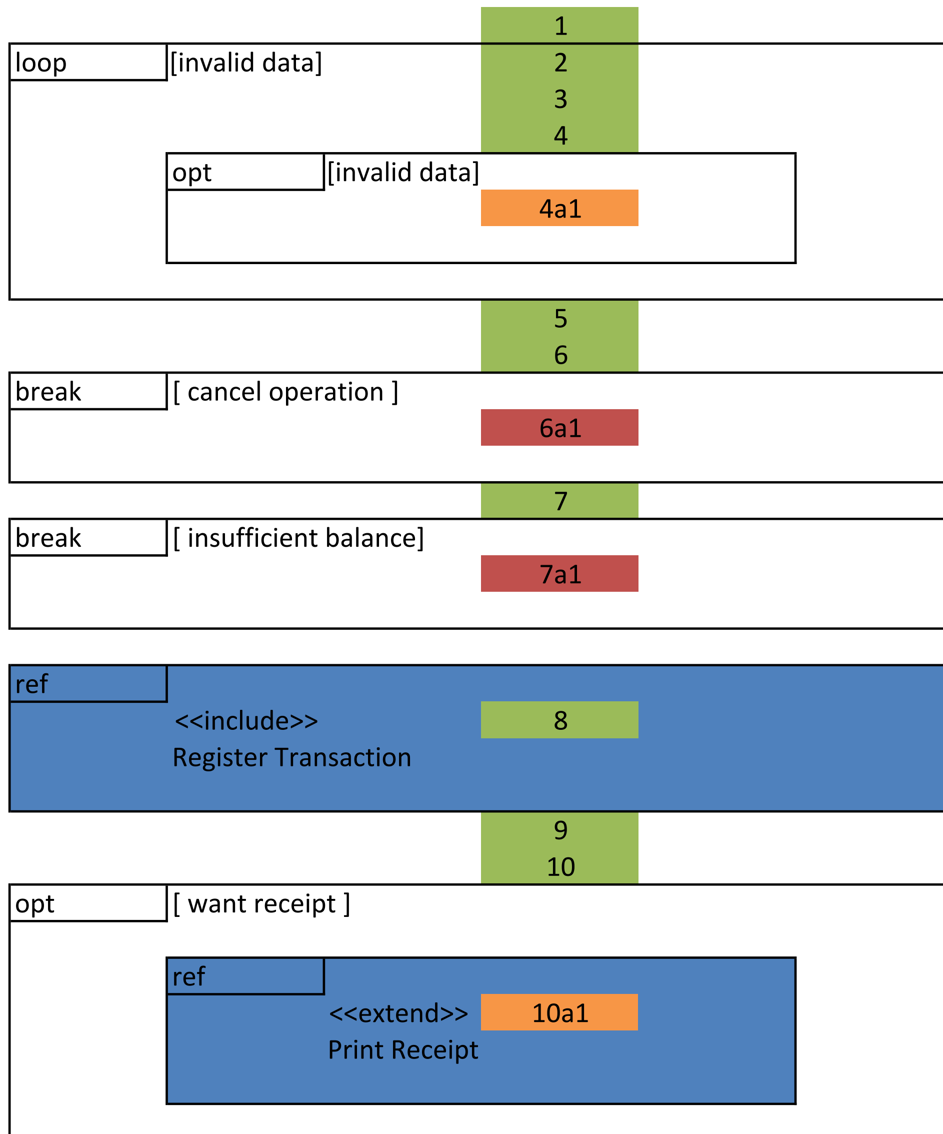


Figure 5.4: Frame schema of the Service Payment use case.

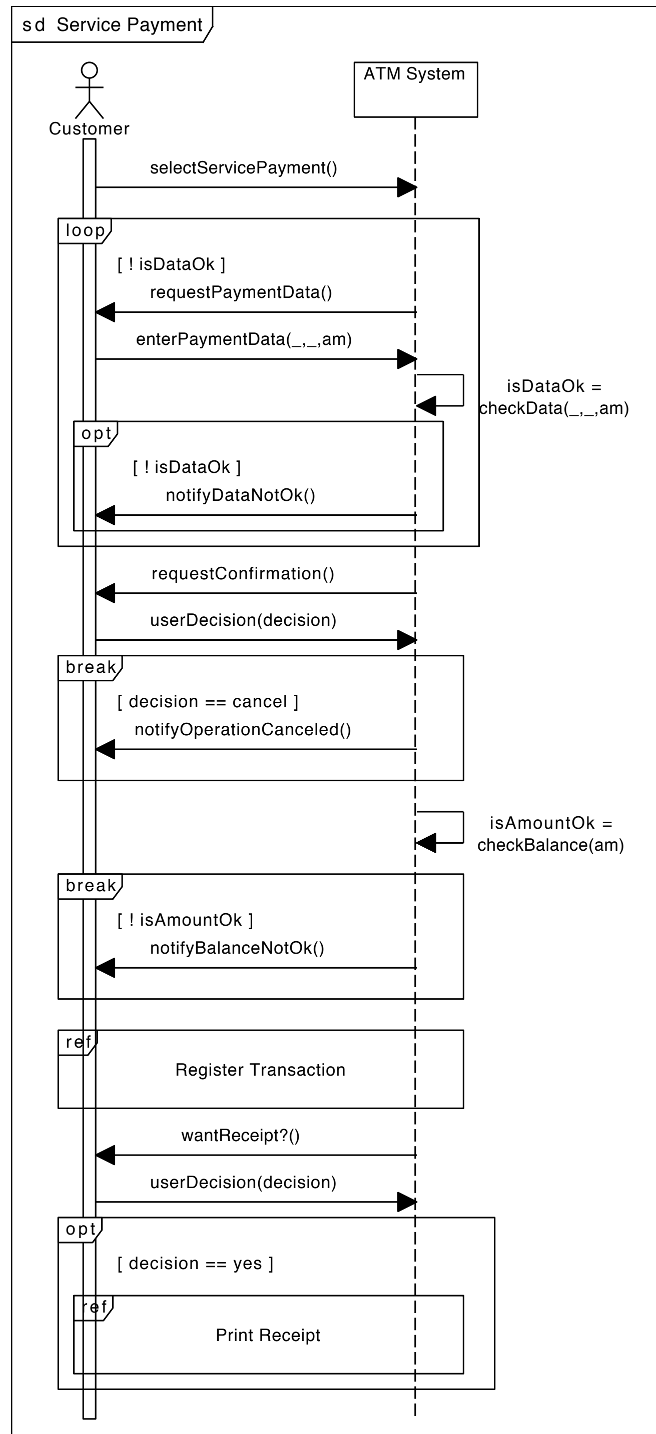


Figure 5.5: System sequence diagram of the Service Payment use case.

rules (Fig. 5.6). We use the same color coding as before to better clarify how the two artefacts match. The action blocks, alternatives and exceptions, inclusions and extensions, and even the action steps that do not fall into any action block are clearly identifiable. For a step-by-step transformation of a use case into an interaction overview diagram see the case study (Section 6).

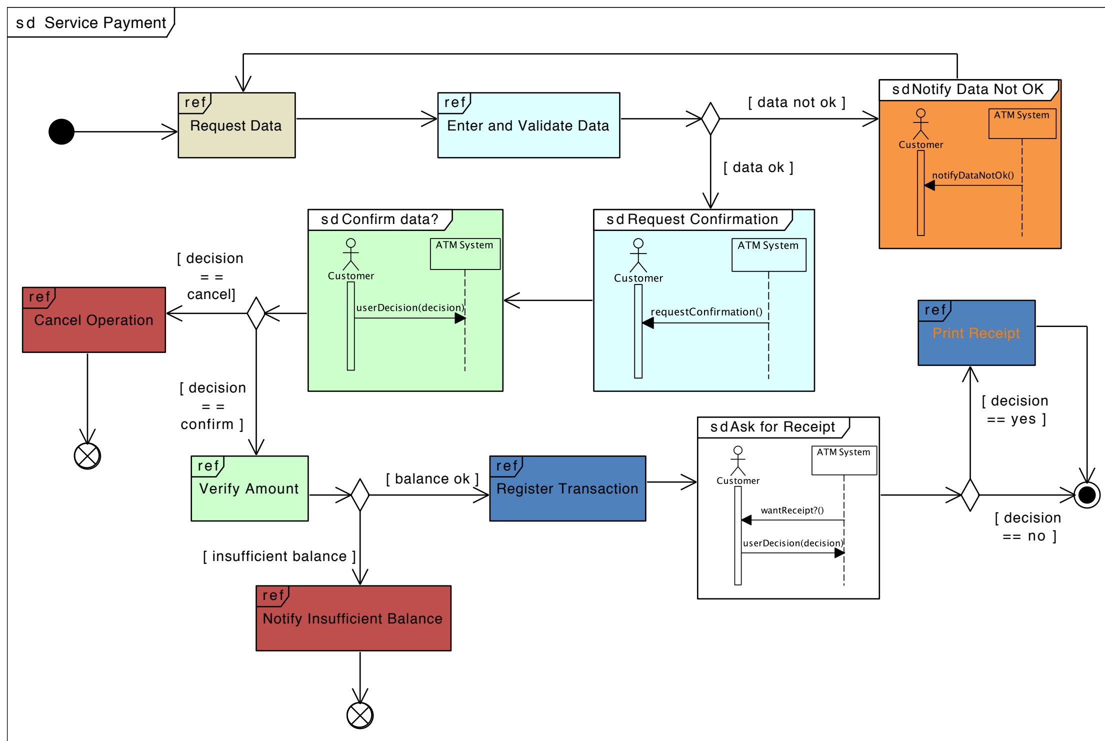


Figure 5.6: Interaction overview diagram of the Service Payment use case.

5.4 Conclusion

In this chapter, we showed the relevance of the role that the transformation rules play in a systematic process of model-driven development. We addressed how this process works generally and the relevance of its main purpose, consistency. Particularly, we presented the transformation rules of use cases to system sequence

diagrams and of use cases to interaction overview diagrams, explaining textually and giving graphical examples for each. Finally, we presented a concise example of the application of the transformation rules.

Chapter 6

Case Study

Having formalized the theoretical transformation rules, it is important to see how they are applied in practical cases. To this end, we provide the case study of an ATM. We model a typical ATM session through some use cases and pick the most diverse one in terms of action steps and action blocks to provide a step-by-step exposition of the transformations into both system sequence and interaction overview diagrams. The models corresponding to the other use cases are presented in Appendix A.

6.1 Domain

ATMs allow clients of financial institutions to make financial transactions without having to go to a house of the institution. With a card and a personal identification number (PIN), clients are allowed to withdraw, deposit or transfer funds, as well as pay for services or check their account balance. Particularly, our use case includes the transfer and withdraw money transactions (Fig. 6.1), which can be performed numerous times during an ATM usage session.

During a session, a customer may perform different transactions. The abstract

act of performing a transaction is represented by the abstract use case *Perform Transaction*. The specific transactions a user may accomplish, like withdrawing or transferring money, specialize this abstract use case.

Since it is not clear at the beginning of a session which transaction will be realized, the *Perform Session* has an include relation with the abstract Perform Transaction use case, meaning that the former will include at some point, the latter's specializations.

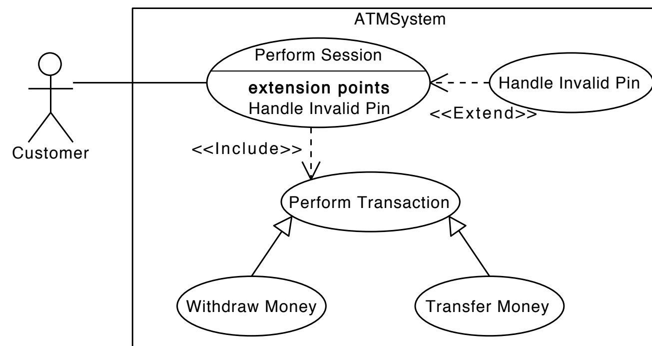


Figure 6.1: Use case diagram of an ATM system.

Every time a customer wants to perform a session he will be asked to enter his PIN. In a typical usage scenario, the user will enter the right PIN. However, should the PIN be wrong, the ATM will employ the proper security mechanisms. Since these mechanisms are used only under certain circumstances, they correspond to an alternative flow and in this case study we have abstracted that alternative into its own use case, *Handle Invalid Pin*, which extends the use case directly associated with the user, Perform Session.

The textual specification of the Perform Session use case given in Table 6.1 is already in the canonical form, a prerequisite for applying the transformation rules. We identify and categorize all action steps and action blocks, as well as all alternatives and corresponding flows.

Table 6.1: Textual specification of the Perform Session use case.

	AB	AS	Narrative
Main Success Scenario	Service	INP	1. Customer inserts card.
		IVAL	2. System verifies card.
		SR	3. System reads card.
		OUT	4. System requests PIN.
	Query	INP	5. Customer enters PIN.
		IVAL	6. System validates PIN.
		OUT	7. System requests a transaction.
			8. *Specialize*: Perform Transaction.
		OUT	9. System asks if the customer wants a receipt.
	Query	UD	10. Customer indicates his choice.
		OUT	11. System asks if the customer wants to perform another transaction.
	Internal	UD	12. Customer indicates his choice.
		SR	13. System ejects card.
		SR	14. System terminates the session.
			15. *Success.*
Exception			2a. **System cannot read card:**
		SR	2a1. System ejects card.
		O	2a2. System notifies customer of the error.
			2a3. *Failure.*
Ext.			6a. **Invalid PIN:**
			6a1. *Extended by*: Handle Invalid PIN.
Alt.			10a. **Customer wants a receipt** (decision = *Yes*)**:**
		SR	10a1. System prints receipt.
			10a2. *Resume.*
Alt.			12a. **Customer wants to perform another transaction** (decision = *Yes*)**:**
			12a1. *Goto* 7.
Alt.			12b. **Customer does not want to perform another transaction** (decision = *No*)**:**
		SR	12b1. System registers the decision.
			12b2. *Resume.*

6.2 From Use Cases to System Sequence Diagrams

In this section, we transform the Perform Session use case into its system sequence diagram counterpart. We divide the aforementioned use case into small portions of behaviour so one can better understand how the transformation rules are used in practice. The transformation of the use case into system sequence diagrams is done one action step at a time. However, since the transformation of non-Choice action steps is straightforward, instead of explaining their transformation one by one, we will, where appropriate, explain them together with other steps. Before starting the transformation, though, to provide a proof of concept, we will present the flow diagram (Fig. 6.2) and frame schema (Fig. 6.3) of the use case we are about to transform. This will show that both approaches indeed coincide in the end result.

6.2.1 Step One

The first step of the use case consists of an input step, whose transformation is trivial. As stated in Section 5.1.1, input steps are mapped into system sequence diagrams by a message from the user to the system. Here, we have called it *insertCard* and passed it a *card* argument (Table 6.2).

Table 6.2: Transformation of Perform Session's step 1 to SSD.

UC Construct	SSD Construct
1. Customer inserts card.	

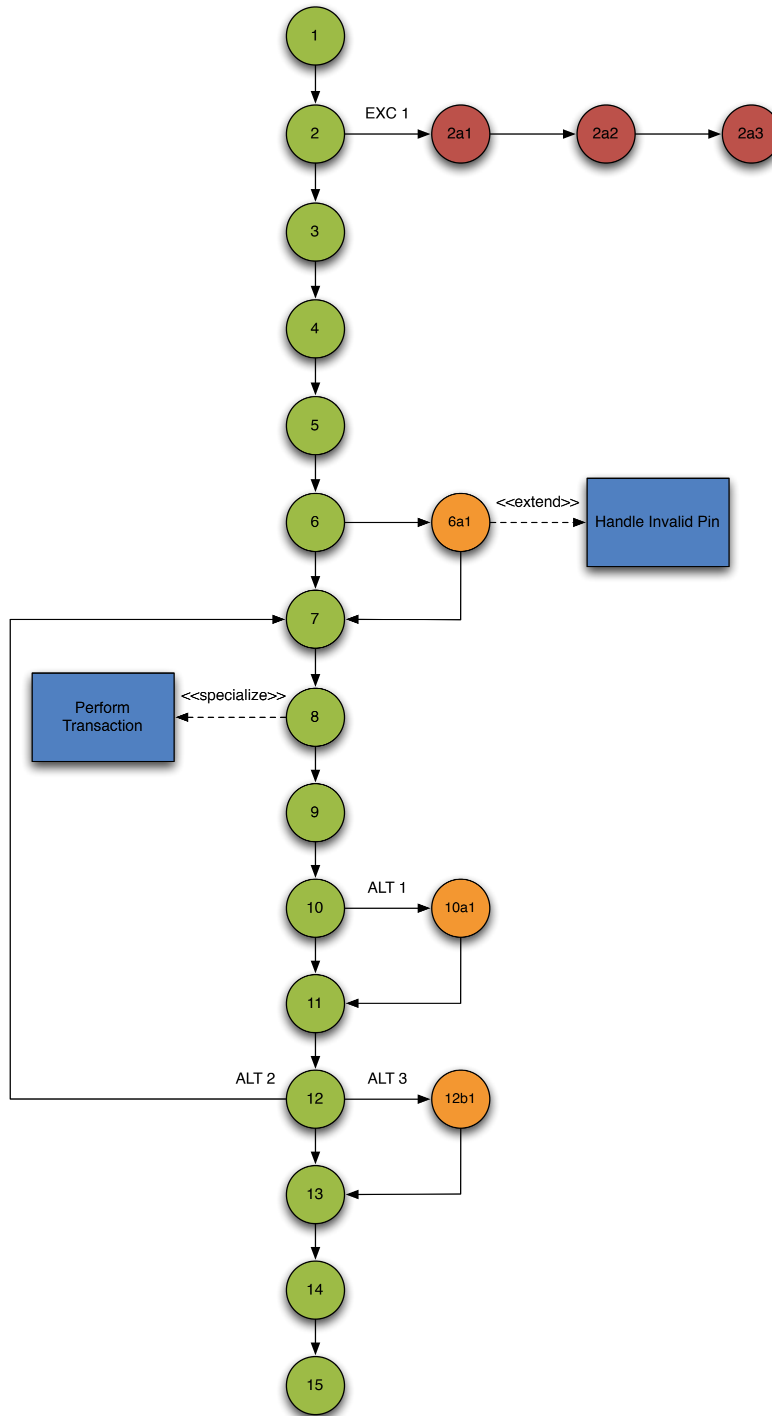


Figure 6.2: Flow diagram of the Perform Session use case.

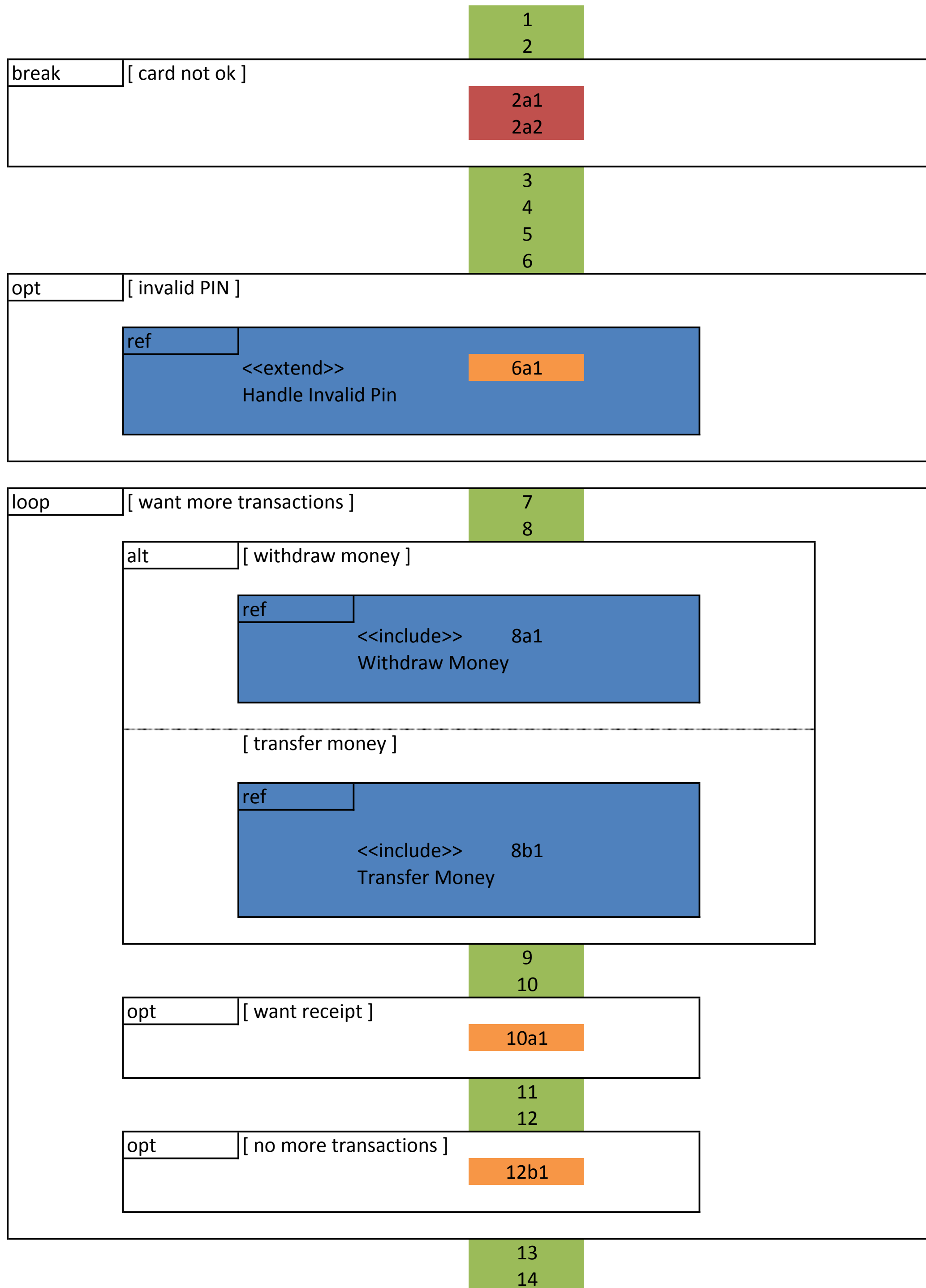


Figure 6.3: Frame schema of the Perform Session use case.

6.2.2 Step Two

The second step of the use case validates the input of the previous one. So, it corresponds to an Input Validation use case construct, which implies an alternative flow. According to the rule summarized in Table 5.5, the validation step itself corresponds to a system self-message in the system sequence diagram.

For the alternative, we need to inspect its last step to decide which frame to use. In this instance, it is a Failure step, signalling the alternative flow models an exception. This means that the suitable option is to use the Break frame. As the alternative flow is composed of two steps, one corresponding to a system responsibility and the other to an output, we placed two messages inside the Break frame. The first being a system self-message mapping the system responsibility step, and the second a message from the system to the customer, reproducing the output (Table 6.3).

Table 6.3: Transformation of Perform Session's step 2 to SSD.

UC Construct	**SSD Construct**
2. System verifies card. 2a. **System cannot read card:** 2a1. System ejects card. 2a2. System notifies customer of the error. 2a3. *Failure.*	isCardOk = checkCard(card) break [! isCardOk] ejectCard() notifyCardNotOk()

6.2.3 Step Three

The use case's steps 3 to 7 are of the same type of action steps as the ones already alluded to. For this reason, we present their transformation together. The transformation of steps 3 (corresponding to message *readCard(card)*), 4 (*requestPin()*),

5 (*enterPin(pin)*) and 7 (*requestTransaction(transOptions)*) is direct since, owed to the type of action step they represent, their transformation is always identical.

The transformation of step 6 is analogous to that of step 2 as they are both input validations. However, there are two differences. The first one is about the type of the step in the alternative flow, which indicates that use case Handle Invalid Pin extends the one we are working on and, therefore, corresponds, in the system sequence diagram, to a Ref frame. The second distinction is the last step of the alternative. Since it is concluded by a Resume step, the flow is of the goto next/resume type. This indicates that, as elucidated by the rule in Section 5.1.2, we use the Opt frame to encompass the alternative flow (Table 6.4).

Table 6.4: Transformation of Perform Session's steps 3 through 7 to SSD.

UC Construct	**SSD Construct**
3. System reads card. 4. System requests PIN. 5. Customer enters PIN. 6. System validates PIN. 7. System requests a transaction. **6a. Invalid PIN:** 6a1. *Extended by*: Handle Invalid PIN.	readCard(card) requestPin() enterPin(pin) isPinOk = checkPin(pin) opt [! isPinOk] ref Handle Invalid Pin requestTransaction(transOptions)

6.2.4 Step Four

Step 8 of the use case corresponds to a Specialize step. This step specifies that a user will be able to perform one among a number of use cases, each of which specializing the use case referenced by the step. This allows specifying textual use cases that will remain unaltered as the number of specializations changes.

"Under the hood", though, a Specialize step corresponds to a user decision. The decision the user has to make is which specializing use case to perform. Each different specialization is enclosed in its own alternative, with an Include step referencing it. In this specific case, for example, there would be two alternatives. One for the *Withdraw Money* use case and another for the *Transfer Money* use case (Table 6.5).

Table 6.5: Concretization of the Specialize step.

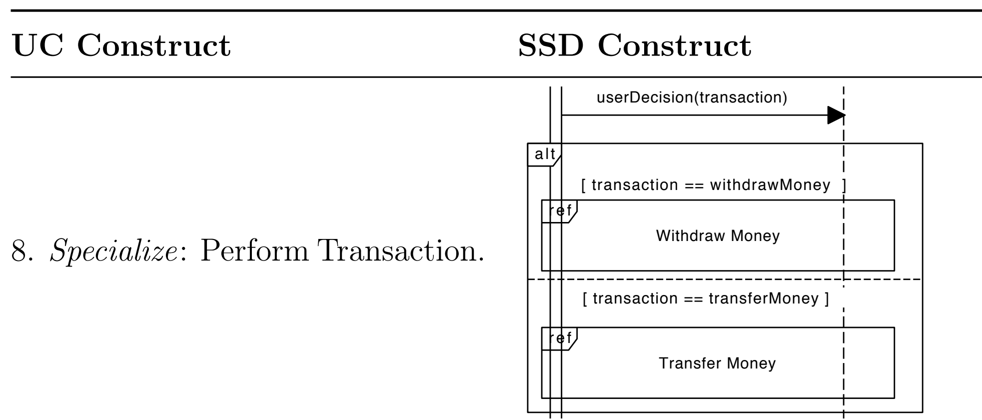
AS	Narrative
	7. ...
UD	8. Customer selects the desired transaction.
	9. ...
	8a. **Customer wants to withdraw money** (decision = *Withdraw Money*)**:**
	8a1. *Include*: Withdraw Money.
	8b. **Customer wants to transfer money** (decision = *Transfer Money*)**:**
	8b1. *Include*: Transfer Money.

The transformation of Specialize steps, into system sequence diagrams, is based on their concretization. Therefore, step 8 of the Perform Session use case is mapped to system sequence diagrams by a user decision message (*userDecision(transaction)*), while its alternatives are enclosed in an Alt frame (Table 6.6).

6.2.5 Step Five

Step 9 of the flow is an output step. It consists of the ATM asking the customer if he wants a receipt and is mapped to system sequence diagrams like all outputs,

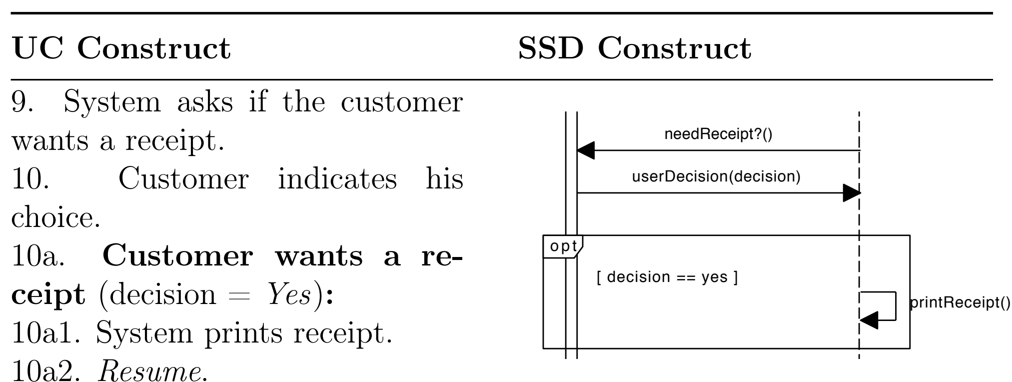
Table 6.6: Transformation of Perform Session's step 8 to SSD.

UC Construct	SSD Construct
8. *Specialize*: Perform Transaction.	

a message from the system to the actor. Here, we called it *needReceipt?()*.

Step 10 is a user decision. Unlike the user decision corresponding to Specialize steps though, the customer choice is binary, being allowed to choose only between "yes" or "no". Since the "no" alternative simply means to continue in the main flow (the textual specification of the alternative would just contain a Resume step) we have omitted it. Thus, only the "yes" alternative was explicitly specified. Hence, and because a Resume step concludes the alternative, we use an Opt frame. Inside the frame, we place a system self-message corresponding to the system responsibility step (Table 6.7).

Table 6.7: Transformation of Perform Session's steps 9 and 10 to SSD.

UC Construct	SSD Construct
9. System asks if the customer wants a receipt. 10. Customer indicates his choice. 10a. **Customer wants a receipt** (decision = *Yes*)**:** 10a1. System prints receipt. 10a2. *Resume*.	

6.2.6 Step Six

This step is analogous to the previous one. There is an output action step followed by a user decision action step. Invariably, the first one corresponds to a message sent by the system to the user. The user decision, however, is still different to the previous ones owed to the nature of the corresponding alternatives.

Similarly to the user decision of step 10, the choices provided in this step are "yes" and "no" as well. The body of the alternative corresponding to the "yes" branch reveals just a Goto step. However, the Goto points to step 7, originating, thus, a cycle. Following the rule for cyclic alternatives, we construct a Loop frame encompassing everything from the message corresponding to step 7 until the message corresponding to the user decision on step 12.

On the other hand, the "no" branch contains a system responsibility action step and ends with a Resume step. Therefore, Opt is the right frame to use and it will contain a system self-message (*setWantMoreTransactions(false)*) corresponding to step 12b1 (Table 6.8).

6.2.7 Step Seven

The last two action steps of the use case are system responsibilities, which are trivially transformed into system self-messages (Table 6.9). The Success step marks the end of the use case and has no special transformation into system sequence diagrams other than concluding the diagram.

Table 6.8: Transformation of Perform Session's steps 11 and 12 to SSD.

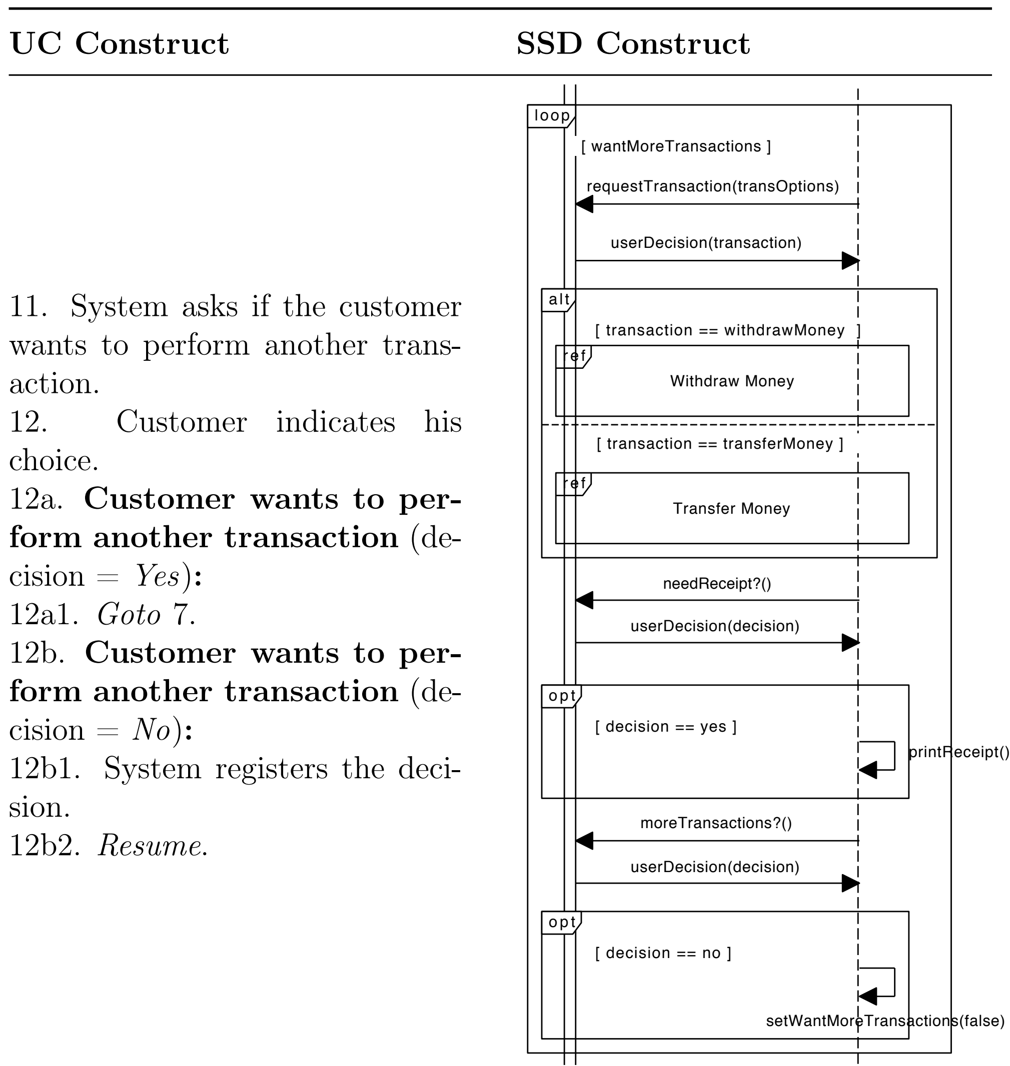
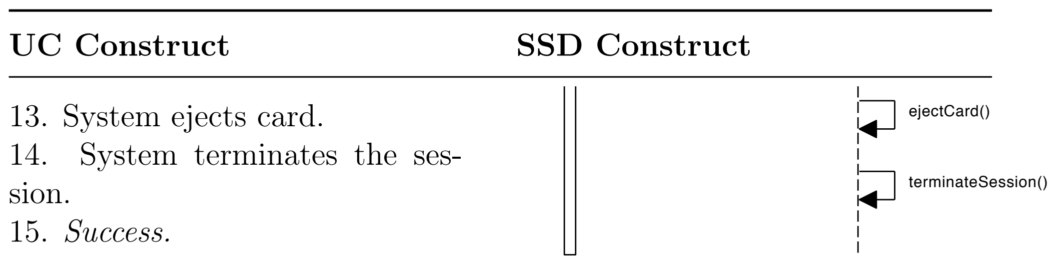
UC Construct	SSD Construct
11. System asks if the customer wants to perform another transaction. 12. Customer indicates his choice. 12a. **Customer wants to perform another transaction** (decision = *Yes*)**:** 12a1. *Goto* 7. 12b. **Customer wants to perform another transaction** (decision = *No*)**:** 12b1. System registers the decision. 12b2. *Resume.*	

Table 6.9: Transformation of Perform Session's steps 13,14 and 15 to SSD

UC Construct	SSD Construct
13. System ejects card. 14. System terminates the session. 15. *Success.*	

6.3 From Use Cases to Interaction Overview Diagrams

The transformation of use cases into interaction overview diagrams is done by transforming one action block at a time. The Perform Session use case contains four action blocks, but since steps 8 and 9 are not part of any of them, the transformation is divided into six parts. One for each action block and another for each one of the steps 8 and 9.

6.3.1 Step One

We start the transformation of the use case to an interaction overview diagram by examining the first action block (steps 1 to 4). Analyzing the steps in the action block, it is possible to see that there is an input validation step. As input validation steps originate alternatives, the rule summarized in Table sh5.14ould be used.

Following the rule, we divide the action block in two and the *Insert and Verify Card* and *Request Pin* Ref frames are created. The former contains the messages corresponding to steps 1 and 2, while the latter contains the other two. Still following the same rule, a decision node is placed after the first Ref frame. The decision node has two outgoing edges, one representing the alternative flow and another representing the continuation of the main flow. We enclosed the messages corresponding to the alternative flow in a third Ref frame, which we designated *Cancel Session*. Since the alternative models an exception, we linked this frame to a flow final node. The branch of the decision node that represents the main flow is linked to the frame containing the second half of the action block (Table 6.10).

Table 6.10: Transformation of Perform Session's first action block to IOD.

UC Construct	SSD Construct
1. Customer inserts card. 2. System verifies card. 3. System reads card. 4. System requests PIN. **2a. System cannot read card:** 2a1. System ejects card. 2a2. System notifies customer of error. 2a3. *Failure.*	ref Insert and Verify Card [card ok] ref Request Pin [card not ok] ref Cancel Session

6.3.2 Step Two

Continuing the transformation, we analyze the second action block (step 5 to 7). Like the first, this action block contains an input validation action step and the same rule should be used. Therefore, we divide the action block in two, placing the messages corresponding to steps 5 and 6 in the first half (the *Enter and Validate Pin* frame) and the message corresponding to the last step on the second half (the *Request Transaction* frame). Analogously to the previous case, these two frames are connected via the main flow branch of the decision node. The alternative flow branch leads to a frame referencing the *Handle Invalid Pin* system sequence diagram. This frame was constructed following the transformation rule for extensions, which is what the alternative flow specified in the use case consists of. Since the last step of the *Handle Invalid Pin* use case is a *Resume* step, the equivalent a *Goto* step pointing to the next step in the main flow, the frame referencing this use case joins the main flow at the *Request Transaction* frame (Table 6.11).

Table 6.11: Transformation of Perform Session's second action block to IOD.

UC Construct	SSD Construct
5. Customer enters PIN. 6. System validates PIN. 7. System requests a transaction. **6a. Invalid PIN:** 6a1. *Extended by*: Handle Invalid PIN.	ref Enter and Validate Pin [pin not ok] ref Handle Invalid Pin [pin ok] sd Request Transaction Customer : ATMSystem requestTransaction()

6.3.3 Step Three

Step 8 of the use case is a *Specialize* step. Since this type of steps correspond to user decisions, the rule for transforming action blocks initiated by a user decision applies. The message corresponding to the user decision is placed in the previous frame of the flow and that frame is succeeded by a decision node. The decision node's outgoing edges each represent one of the options available to the user. Here, the customer may choose between transferring money or withdrawing money (Table 6.12).

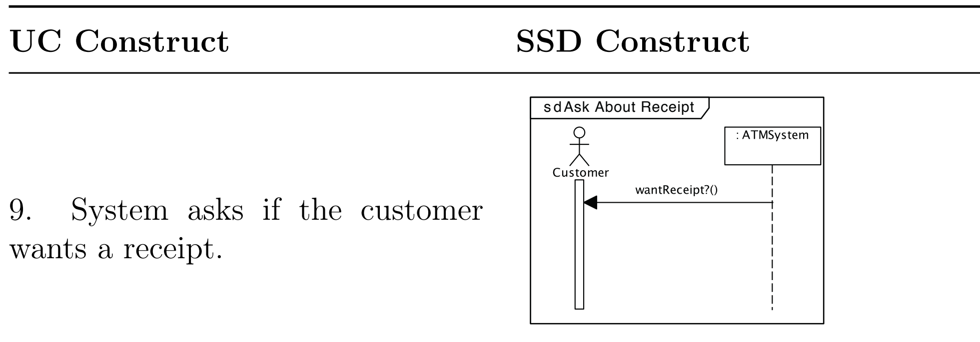
Table 6.12: Transformation of Perform Session's step 8 to IOD.

UC Construct	SSD Construct
8. *Specialize*: Perform Transaction.	sd Request Transaction Customer : ATMSystem requestTransaction() userDecision(transaction) [transaction == TransferMoney] ref Transfer Money [transaction == WithdrawMoney] ref Withdraw Money

6.3.4 Step Four

After each transaction, the customer is provided the choice to print a receipt, represented by step 9 of the use case. This step does not belong to any action block and, therefore, is simply depicted as an embedded sequence diagram by itself (Table 6.13).

Table 6.13: Transformation of Perform Session's step 9 to IOD.

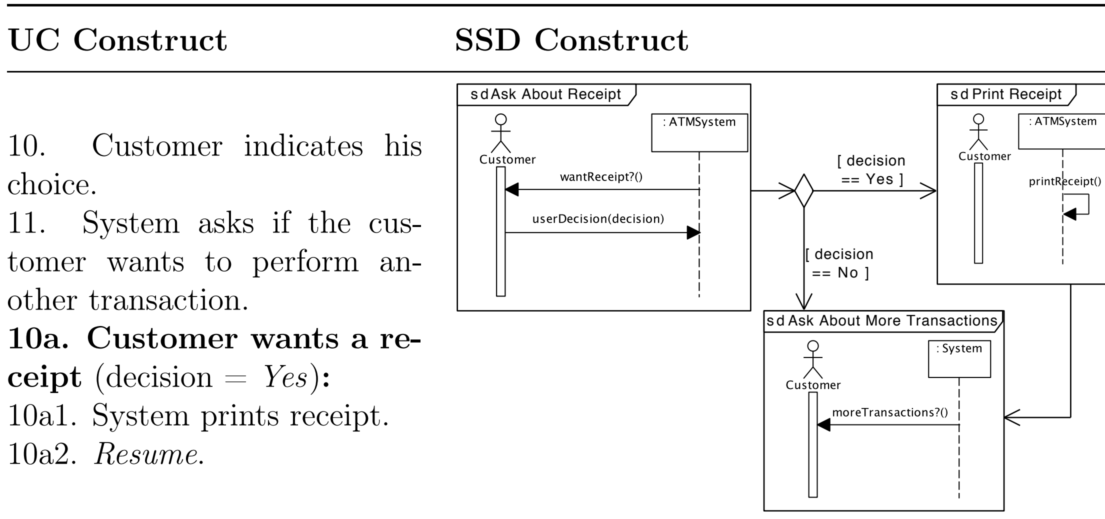
UC Construct	SSD Construct
9. System asks if the customer wants a receipt.	

6.3.5 Step Five

The following action block comprises steps 10 and 11. As step 10, the first step of the action block, is a user decision, the suitable rule to use is the one used before for step 8; i.e., the rule for action blocks initiated by user decisions. As before, the message corresponding to the user decision is placed in the previous frame. After which there is a decision node. Again, one of the options the user can take keeps him in main flow while the other leads him to an alternative flow. The main flow branch is connected to the frame containing the remaining messages of the action block; here, the message corresponding to step 12 (*moreTransactions?()*). The alternative flow related to this action block contains a single action step, which is depicted inside the *Print Receipt* frame in Table 6.14, and is concluded by a *Resume* step. This means that the alternative flow rejoins the main flow

at the step after the user decision, which in this case means the *Ask About More Transactions* frame.

Table 6.14: Transformation of Perform Session's third action block to IOD.

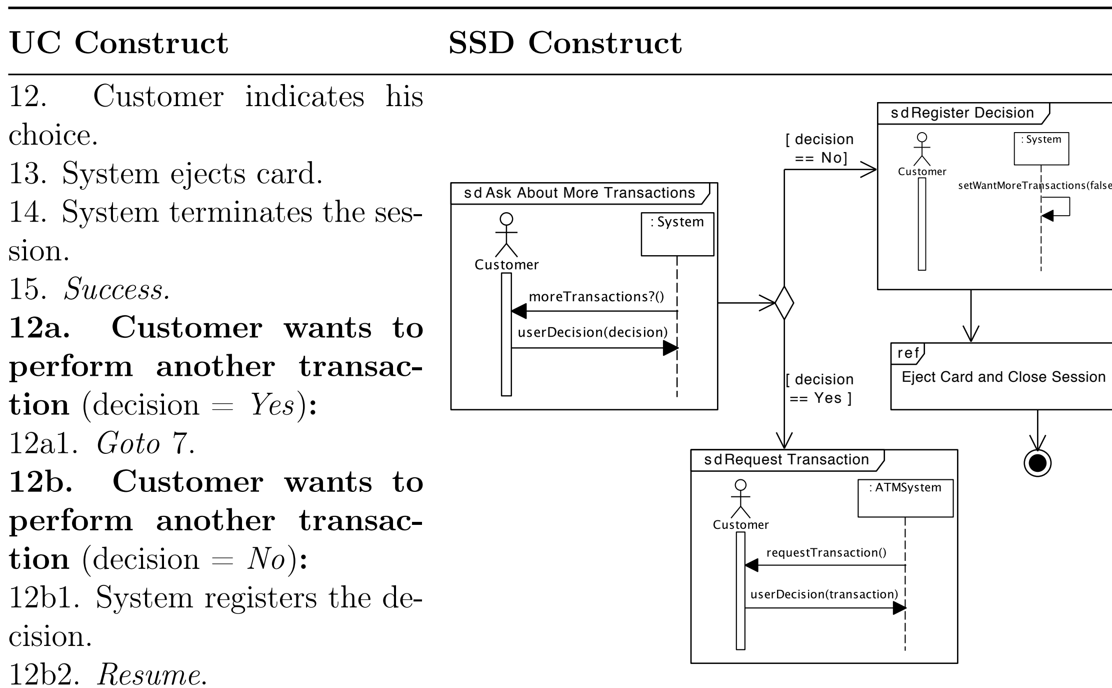
UC Construct	**SSD Construct**
10. Customer indicates his choice. 11. System asks if the customer wants to perform another transaction. **10a. Customer wants a receipt** (decision = *Yes*)**:** 10a1. System prints receipt. 10a2. *Resume.*	

6.3.6 Step Six

The last action block of the use case (steps 12 to 14) again begins with a user decision. The transformation is analogous. First, the user decision message is put on the previous frame of the flow, then a decision node is included in the diagram follows. This time, however, there are two distinct alternative flows branching from the decision node. The alternative where the customer chooses to perform more transactions contains a single step, a Goto. The rule used in this case is the same we used before for flows that ended with a Resume step; i.e., an arrow leading to the frame where the message corresponding to the action step pointed by the Goto is. Here, choosing to perform more transactions leads to the *Request Transaction* frame. If, otherwise, the user decides not to perform more transactions, the decision node leads to the *Register Decision* frame, which is the frame that contains the corresponding alternative flow. Since that alternative flow is concluded by a

Resume step, the respective frame joins the main flow, which corresponds to the second part of the action block, the *Eject Card and Close Session* Ref frame. After which the use case terminates with success (Table 6.15).

Table 6.15: Transformation of Perform Session's fourth action block to IOD.

UC Construct	**SSD Construct**
12. Customer indicates his choice. 13. System ejects card. 14. System terminates the session. 15. *Success.* **12a. Customer wants to perform another transaction** (decision = *Yes*): 12a1. *Goto* 7. **12b. Customer wants to perform another transaction** (decision = *No*): 12b1. System registers the decision. 12b2. *Resume.*	

6.4 Consistency Verification

To verify the consistency of the transformed models, each one has been coded in Alloy. For this purpose, the entities that allow instances have been made abstract so that we can control which instances the Alloy Analyzer creates. Had we not done this, besides our manually created instances which are unique owed to the `one` multiplicity keyword used in their signature, the Alloy Analyzer would create random ones by itself.

The code for each instance is excessively long, so here we will just present some

snippets. The following snippet shows how the entities corresponding to the use case model, actor and use cases were instanced in Alloy:

```
one sig ATMUseCaseModel extends UseCaseModel {}
one sig Customer extends User {}
one sig PerformSession, PerformTransaction, HandleInvalidPIN,
    WithdrawMoney, TransferMoney extends UseCase {}
```

Instantiating an entity is done analogously for all entities. The relations between entities are modeled via facts. The following snippet, taken from the Perform Session interaction overview diagram instance, depicts a fact stating the sender and receiver of some messages:

```
fact messages {
    source = requestTransaction -> ATMSystem +
             printReceipt -> ATMSystem +
             moreTransactions -> ATMSystem

    target = requestTransaction -> Customer +
             printReceipt -> ATMSystem +
             moreTransactions -> Customer
}
```

Manually creating the instances in Alloy is a time-consuming and error-prone process. Also, debugging complex and long Alloy models, as the instances tend to be, is difficult since Alloy only states whether a model is consistent or not and does not provide an easy way to find the error and fix it. It is not uncommon that inconsistencies are caused by a faulty description of the instances rather than by faulty model transformations.

6.5 Conclusion

In this chapter, we have applied the transformation rules provided in Chapter 5 in a practical context. We picked the domain of ATMs and created use cases that model a typical usage session. The chosen use case for the step by step transformation, both into system sequence and interaction overview diagrams, was the one that modeled a session as whole because it was the richest in terms of the different action steps and action blocks contained.

Chapter 7

Conclusion and Future Work

In this thesis we have presented a canonical form for writing textual use cases, rules for the systematic transformation of use cases to system sequence diagrams and interaction overview diagrams and a formal mechanism for verifying the correction of these transformations.

The Unified Modeling Language (UML) specification provides meta-models and Object Constraint Language (OCL)-based rules for the well-formedness of its diagrams in an attempt to imbue some degree of formalization into the language. However, the textual representation of use cases, a resource commonly used in practice, is not touched upon by the specification. Hence, there is not a standard way to textually specify use cases. The canonical form of writing use cases proposed in this thesis attempts to provide a methodical and structured way to specify use cases which, furthermore, lays down the basis for the progressive design of the software system.

Using this basis as a solid foundation upon which to build the rest of the computer system, a set of transformation rules was developed to continue the systematic construction of the system. The transformation rules use the meta-data existent in the use cases and map it to both system sequence diagrams and

interaction overview diagrams. The application of these rules is systematic and provides a way of methodically and unambiguously construct the aforementioned diagrams. This means that if two independent modelers were given the same use case, they should arrive to the same system sequence diagram and interaction overview diagram. We believe this level of determinism is crucial to eliminate many of the errors that arise during the design phase of a software project.

Moreover, since UML is only semi-formal, we have used Alloy to develop fully formalized meta-models of use case diagrams and their textual representations, system sequence diagrams, and interaction overview diagrams. Using this formal basis, we also developed a mechanism for the verification of the transformation process. Thus, it is possible for a modeler to confirm the transformation is accurate.

However, as in any project, there is always room for improvement either by adding new features or perfecting the existing ones. One way the current work could be improved would be by implementing a module to automatically transform use cases to system sequence diagrams and interaction overview diagrams. This feature would save time and possibly avoid transformation errors, considering that the manual transformations, although systematic, are not error safe.

Also, as mentioned before, creating the Alloy instances to be analyzed is a time-consuming and error prone process. An external tool with capacity to create the UML models and automatically generate the corresponding Alloy code would make this process a lot more agile and robust.

The application of this work in more practical contexts could also uncover some other limitations. Thus, empirical studies of projects using this methodology are encouraged and could result in the discovery of more transformation rules or modifications on the existing ones.

Another line of work would be to add more UML diagrams to the process. Particularly, refined sequence diagrams and posteriorly class diagrams would be a

logical next step to making the framework more thorough and comprehensive.

Bibliography

[ABGR10] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and System Modeling*, 9(1):69–86, 2010.

[AJI07] Jesús Manuel Almendros-Jiménez and Luis Iribarne. Describing use-case relationships with sequence diagrams. *Comput. J.*, 50(1):116–128, 2007.

[AM01] Frank Armour and Granville Miller. *Advanced use case modeling: software systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[ASP04] Nuno Amálio, Jj Susan Stepney, and Fiona Polack. Formal proof from UML models. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2004.

[Bar06] F. Barajas. A formal model for a requirements engineering tool. In *Proceedings of First Alloy Workshop'06*, 2006.

[BG01] Jean Bézivin and Olivier Gerbé. Towards a precise definition of the OMG/MDA framework. In *ASE*, pages 273–280. IEEE Computer Society, 2001.

[BMP+02] Jean-Paul Bodeveix, Thierry Millan, Christian Percebois, Christophe Le Camus, Pierre Bazex, and Louis Feraud. Extending OCL for verifying UML models consistency. In Kuzniarz et al. [KHRS02], pages 75–90.

[Boo95] Grady Booch. *Object-oriented analysis and design with applications (2. ed.)*. Benjamin/Cummings series in object-oriented software engineering. Addison-Wesley, 1995.

[BRJ05] Grady Booch, James Rumbaugh, and Ivar Jacobson. *Unified Modeling Language User Guide, The*. Addison-Wesley Professional, 2 edition, 2005.

[BV06] Umesh Bellur and V. Vallieswaran. On OO design consistency in iterative development. In *ITNG*, pages 46–51. IEEE Computer Society, 2006.

[Coc00] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 2000.

[CPC+04] Dan Chiorean, Mihai Pasca, Adrian Cârcu, Cristian Botiza, and Sorin Moldovan. Ensuring UML models consistency using the OCL environment. *Electr. Notes Theor. Comput. Sci.*, 102:99–110, 2004.

[CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.

[EB04] Maged Elaasar and Lionel Claude Briand. An overview of UML consistency management. Technical report, Department of Systems and Computer Engineering, Carleton University, 2004.

[EEPT10] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[Fow03] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[FSKdR05] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem P. de Roever. 29 new unclarities in the semantics of UML 2.0 state machines. In Kung-Kiu Lau and Richard Banach, editors, *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2005.

[GCP+05] José Daniel García, Jesús Carretero, José María Pérez, Félix García Carballeira, and Rosa Filgueira. Specifying use case behavior with interaction models. *Journal of Object Technology*, 4(9):143–159, 2005.

[GKZ09] Sabine Kuske Jjand Martin Gogolla, Hans-Jörg Kreowski, and Paul Ziemann. Towards an integrated graph-based semantics for UML. *Software and System Modeling*, 8(3):403–422, 2009.

[HA09] Johan Helldahl and Usman Ashraf. Use case explorer - a use case tool. Master's thesis, Chalmers University of Technology, 2009.

[Hel05] Rogardt Heldal. Use cases are more than system operations. In *2nd International Workshop on Use Case Modelling (WUsCaM-2005)*, 2005.

[Hol04] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[Hru97] Peter Hruschka. UML: A Curse or a Blessing for the OO Community. *American Programmer*, 10(3), 1997.

[IIS+10] Noraini Ibrahim, Rosziati Ibrahim, Mohd Zainuri Saringat, Dzahar Mansor, and Tutut Herawan. On well-formedness rules for UML use case diagram. In Fu Lee Wang, Zhiguo Gong, Xiangfeng Luo, and Jingsheng Lei, editors, *WISM*, volume 6318 of *Lecture Notes in Computer Science*, pages 432–439. Springer, 2010.

[Jac00] Daniel Jackson. Automating first-order relational logic. In *SIGSOFT FSE*, pages 130–139, 2000.

[Jac02] D. Jackson. Micromodels of Software: Lightweight Modelling and Analysis With Alloy. Technical report, Software Design Group. MIT Lab for Computer Science, 2002.

[Jac06] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[Jac12] Daniel Jackson. Alloy analyzer website. http://alloy.mit.edu/, 2012.

[JCJÖ92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Övergaard. *Object-oriented software engineering - a use case driven approach*. Addison-Wesley, 1992.

[JW96] Daniel Jackson and Jeannette Wing. Formal methods light: Lightweight formal methods. *Computer*, 29(4):21–22, April 1996.

[KG03] Daryl Kulak and Eamonn Guiney. *Use Cases: Requirements in Context*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003.

[KHR+03] Ludwik Kuzniarz, Zbigniew Huzar, Gianna Reggio, Jean Sourrouille, and Miroslaw Staron. Workshop on consistency problems in UML-based software development II. Blekinge Institute of Technology, 2003.

[KHR+04] Ludwik Kuzniarz, Zbigniew Huzar, Gianna Reggio, Jean Sourrouille, and Miroslaw Staron. Workshop on consistency problems in UML-based software development III. Blekinge Institute of Technology, 2004.

[KHRS02] Ludwik Kuzniarz, Zbigniew Huzar, Gianna Reggio, and Jean Sourrouille. Workshop on consistency problems in UML-based software development I. Blekinge Institute of Technology, 2002.

[KM08] Pierre Kelsen and Qin Ma. A lightweight approach for defining the formal semantics of a modeling language. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *MoDELS*, volume 5301 of *Lecture Notes in Computer Science*, pages 690–704. Springer, 2008.

[Kne97] Ralf Kneuper. Limits of formal methods. *Formal Asp. Comput.*, 9(4):379–394, 1997.

[Kru03] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction*

to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall PTR, Upper Saddle River, NJ, USA, 3rd edition, 2004.

[Li00] Liwu Li. Translating use cases to sequence diagrams. In *ASE*, pages 293–296, 2000.

[MBC05] Johan Muskens, Reinder J. Bril, and Michel R. V. Chaudron. Generalizing consistency checking between software views. In *WICSA*, pages 169–180. IEEE Computer Society, 2005.

[MOW03] Pierre Metz, John O'Brien, and Wolfgang Weber. Specifying use case interaction: Types of alternative courses. *Journal of Object Technology*, 2(2):111–131, 2003.

[MS09] PAJ Mason and S Supsrisupachai. Paraphrasing use case descriptions and sequence diagrams: An approach with tool support. In *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, 2009. ECTI-CON 2009. 6th International Conference on*, volume 2, pages 722–725. IEEE, 2009.

[NYM+10] Akie Nimiya, Tomoyuki Yokogawa, Hisashi Miyazaki, Sousuke Amasaki, Yoichiro Sato, and Michiyoshi Hayas. Model checking consistency of UML diagrams using Alloy. In *WASET*, pages 547–550, 2010.

[OMG10] Object constraint language. Technical report, Object Management Group, Framingham, MA, February 2010. Version 2.2.

[OMG11] OMG unified modeling language (OMG UML), superstructure. Technical report, Object Management Group, Framingham, MA, August 2011. Version 2.4.1.

[RBP+91] James E. Rumbaugh, Michael R. Blaha, William J. Premerlani, Frederick Eddy, and William E. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[SAB09] Seyyed M. A. Shah, Kyriakos Anastasakis, and Behzad Bordbar. From UML to Alloy and back again. In Sudipto Ghosh, editor, *MoDELS Workshops*, volume 6002 of *Lecture Notes in Computer Science*, pages 158–171. Springer, 2009.

[SB05] Jocelyn Simmonds and M. Cecilia Bastarrica. A tool for automatic UML model consistency checking. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *ASE*, pages 431–432. ACM, 2005.

[SCK09] Daniel SinnigJj, Patrice Chalin, and Ferhat Khendek. LTS semantics for use case models. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 365–370. ACM, 2009.

[SG99] Anthony J. H. Simons and Ian Graham. 30 things that go wrong in object modelling with UML 1.3. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 16, pages 221–242. Kluwer, Dordrecht, 1999.

[SHH07] Laura Mendez Segundo, Rodolfo Romero Herrera, and K. Yeni Perez Herrera. UML sequence diagram generator system from use case description using natural language. In *Proceedings of the Electronics, Robotics and Automotive Mechanics Conference*, pages 360–363, Washington, DC, USA, 2007. IEEE Computer Society.

[Sim99] Anthony J. H. Simons. Use cases considered harmful. In *TOOLS (29)*, pages 194–203. IEEE Computer Society, 1999.

[SP06] Perdita Stevens and Rob Pooley. *Using Uml: Software Engineering with Objects and Components*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2006.

[WD96] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[WGN03] R. Wagner, H. Giese, and U. Nickel. A plug-in for flexible and incremental consistency management. In Kuzniarz et al. [KHR+03], pages 78–85.

[WKKP05] Clay Williams, Matthew Kaplan, Tim Klinger, and Amit M. Paradkar. Toward engineered, useful use cases. *Journal of Object Technology*, 4(6):45–57, 2005.

[YBL10] Tao Yue, L.C. Briand, and Yvan Labiche. Automatically deriving UML sequence diagrams from use cases. Technical report, Simula Research Laboratory, 2010.

[ZLQ06] Xiangpeng Zhao, Quan Long, and Zongyan Qiu. Model checking dynamic UML consistency. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 440–459. Springer, 2006.

Appendix A

Case Study Diagrams

A.1 Textual Use Cases

Table A.1: Textual specification of the Transfer Money use case.

	AB	AS	Narrative
M. S. Scenario	Query	INP	1. Customer chooses to transfer money.
		OUT	2. System requests the type of account to transfer from, an account to transfer to, and the amount of money to transfer.
	Validation	INP	3. Customer enters requested data.
		IVAL	4. System verifies if the amount is less than or equal to the account balance.
			5. *Resume.*
Exception			4a. **Amount > Balance:**
		OUT	4a1. System notifies customer of insuficient balance.
		SR	4a2. System ejects card.
			4a3. *Failure.*

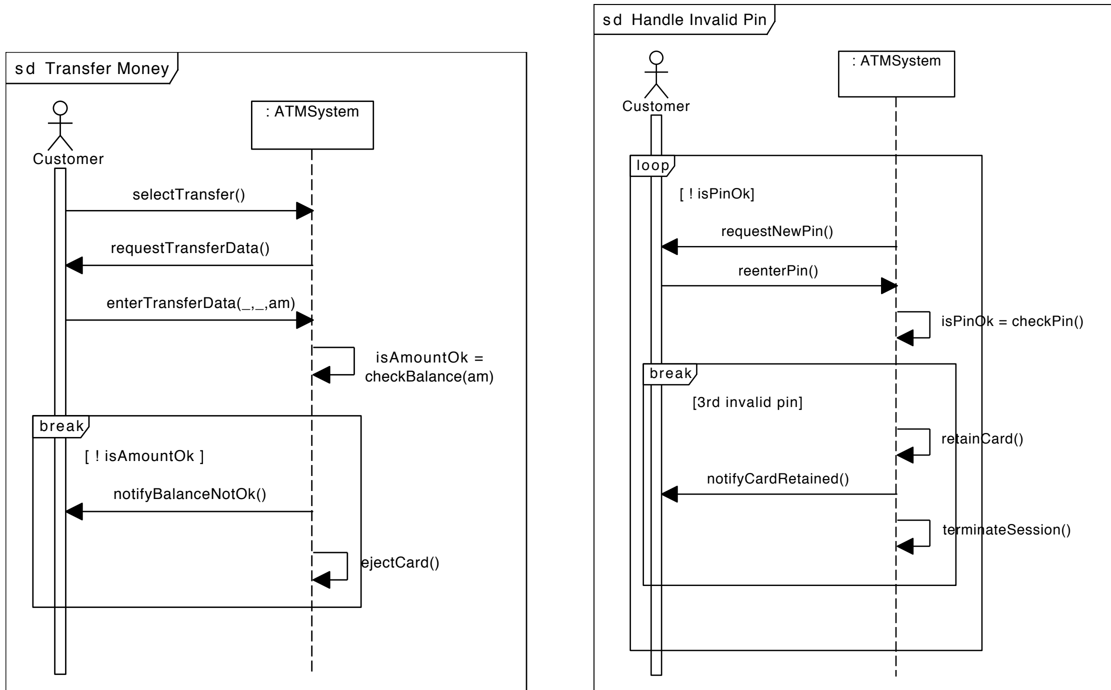
Table A.2: Textual specification of the Handle Invalid Pin use case.

	AB	AS	Narrative
M.S.S.		OUT	1. System requests PIN again.
	Val.	INP	2. Customer reenters PIN.
	Val.	IVAL	3. System validates PIN.
			4. *Resume.*
Alt.			3a. **Second invalid PIN submission:**
			3a1. *Goto 1.*
Exception			3b. **Third invalid PIN submission:**
		SR	3b1. System retains card.
		OUT	3b2. System notifies customer.
		SR	3b3. System terminates session.
			3b4. *Failure.*

Table A.3: Textual specification of the Withdraw Money use case.

	AB	AS	Narrative
Main Success Scenario	Query	INP	1. Customer chooses to withdraw money.
		OUT	2. System asks for a type of account to withdraw money from.
	Query	INP	3. Customer selects the type of account.
		OUT	4. System request the amount of money to withdraw.
	Internal	INP	5. Customer enters the amount of money to withdraw.
		IVAL	6. System verifies if the amount is less than or equal to the account balance.
		SC	7. System verifies if it has sufficient money on hand.
		SR	8. System dispenses the cash.
			9. *Resume.*
Exception			6a. **Amount > Balance:**
		OUT	6a1. System notifies customer of insufficient balance.
		SR	6a2. System ejects card.
			6a3. *Failure.*
Alternative			7a. **System does not have sufficient money on hand:**
		OUT	7a1. System notifies customer there is not enough money on hand.
		OUT	7a1. System asks customer to enter a smaller amount.
			7a2. *Goto 5.*

A.2 System Sequence Diagrams



(a) System sequence diagram of the Transfer Money use case.

(b) System sequence diagram of the Handle Invalid Pin use case.

Figure A.1: System sequence diagrams of the Transfer Money and Handle Invalid PIN use cases.

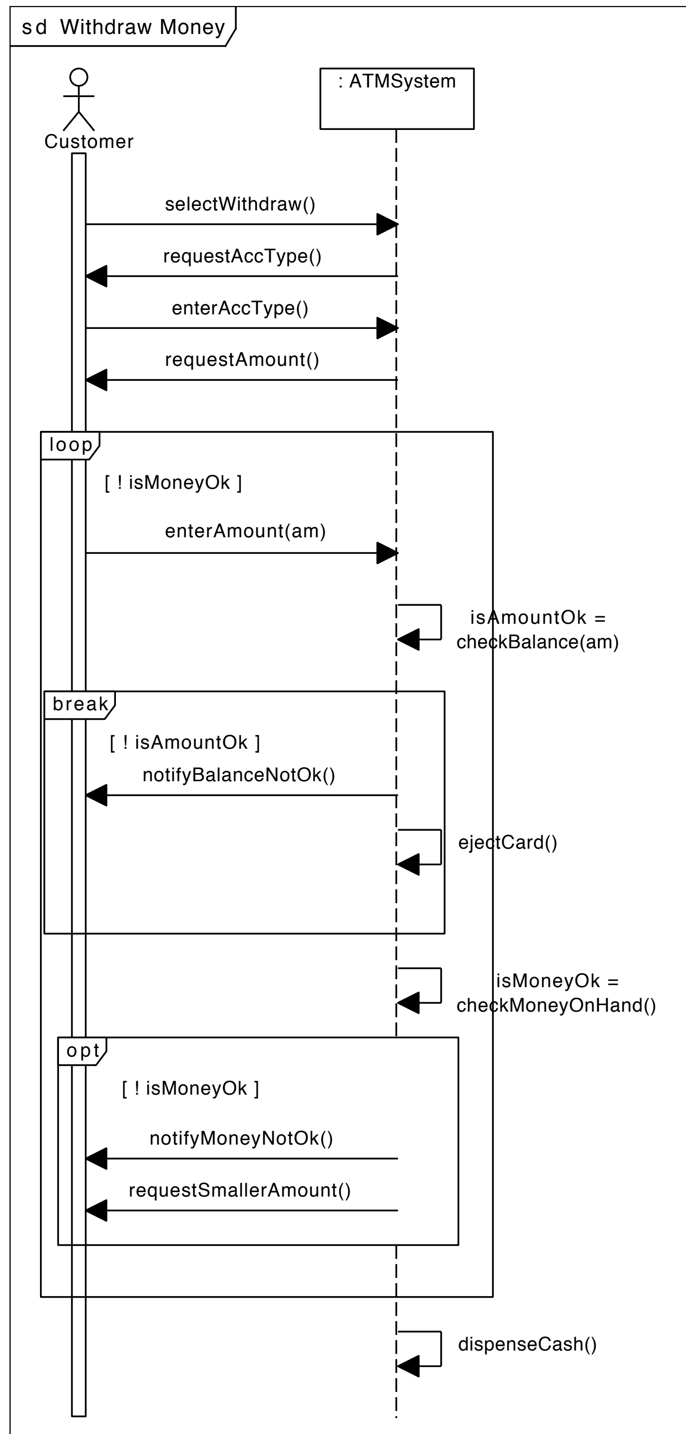


Figure A.2: System sequence diagram of the Withdraw Money use case.

A.3 Interaction Overview Diagrams

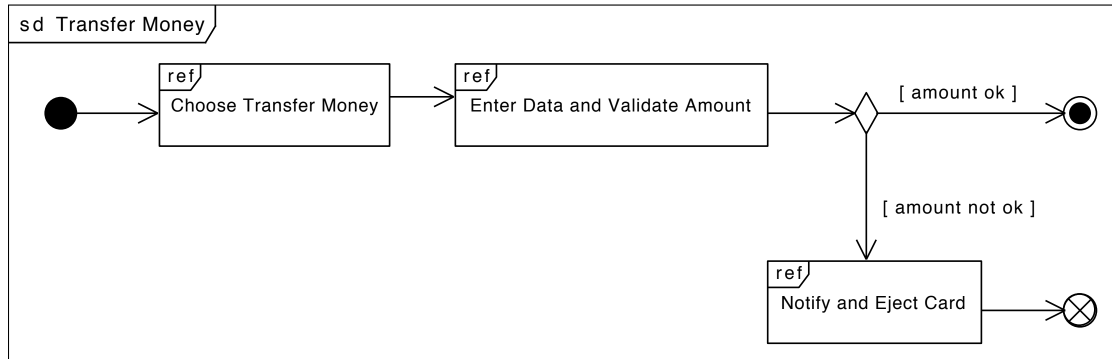


Figure A.3: Interaction overview diagram of the Transfer Money use case.

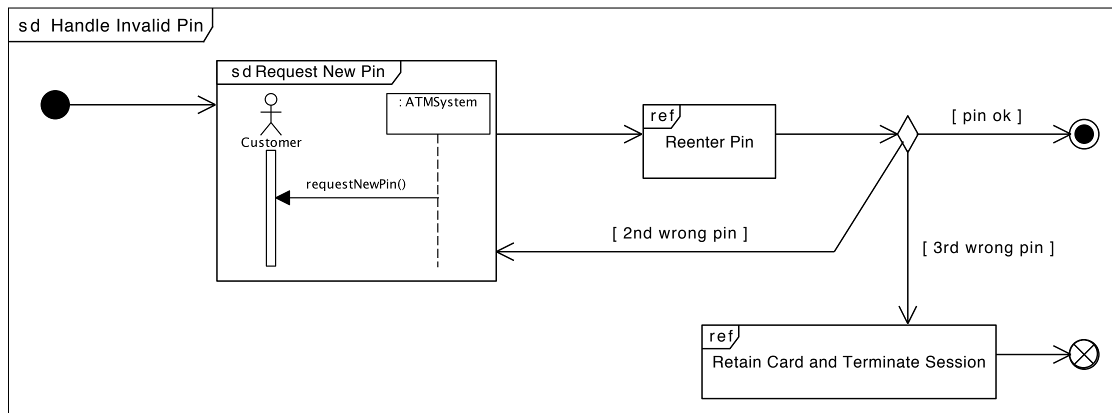


Figure A.4: Interaction overview diagram of the Handle Invalid Pin use case.

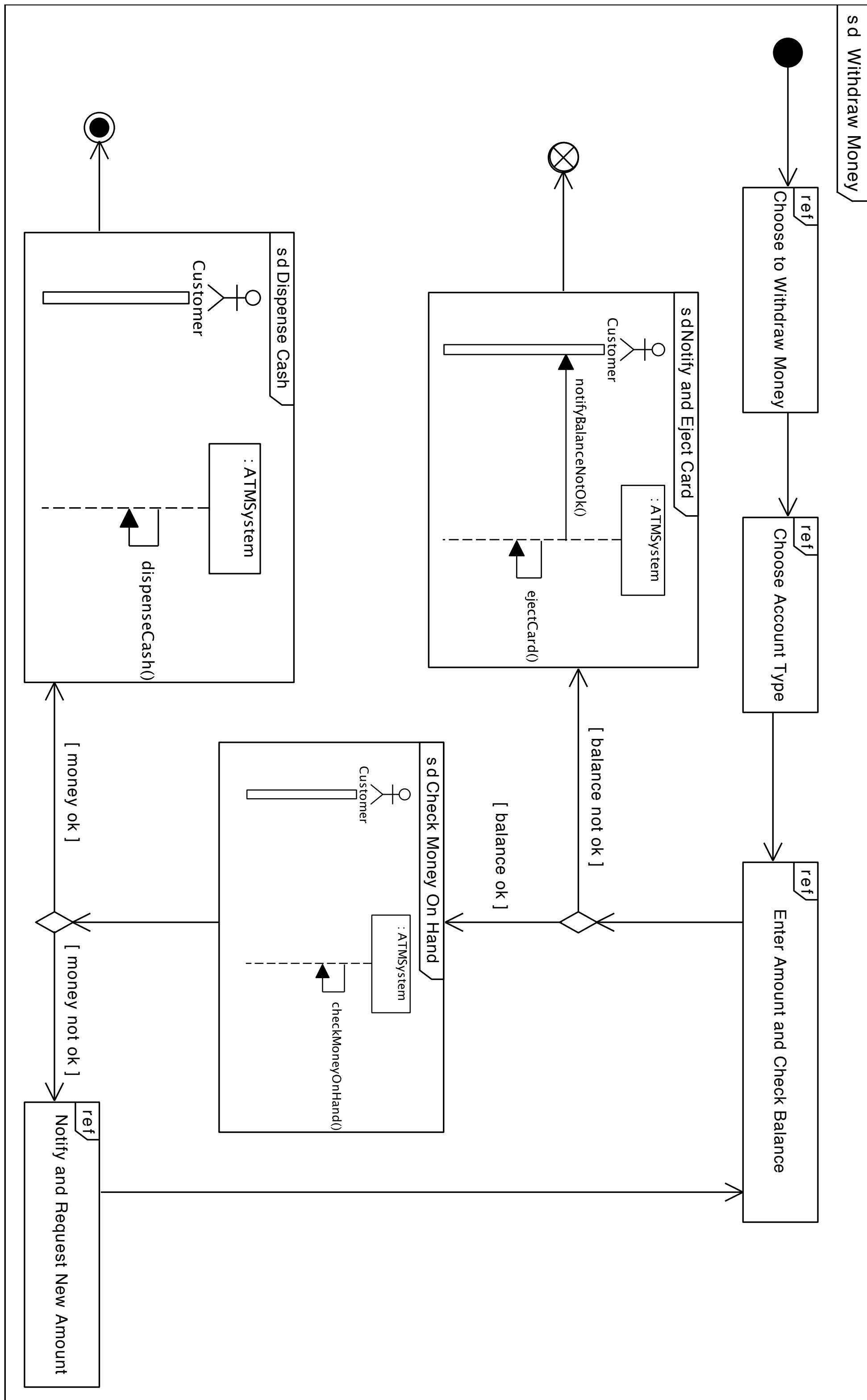


Figure A.5: Interaction overview diagram of the Withdraw Money use case.