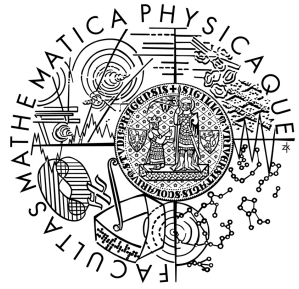
CHARLES UNIVERSITY IN PRAGUE
FACULTY OF MATHEMATICS AND PHYSICS

DOCTORAL THESIS



Vladimír Mencl

Use Cases: Behavior Assembly, Behavior Composition and Reasoning

Department of Software Engineering
Advisor: Prof. Ing. František Plášil, DrSc.

Abstract

Title: Use Cases: Behavior Assembly, Behavior Composition and Reasoning
Author: Mgr. Vladimír Mencl
email: vladimir.mencl@mff.cuni.cz
phone: +420 2 2191 4232
Department: Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic
Advisor: Prof. Ing. František Plášil, DrSc.
email: plasil@nenya.ms.mff.cuni.cz
phone: +420 2 2191 4266
Mailing address (both Author and Advisor):
Dept. of SW Engineering, Charles University in Prague
Malostranské nám. 25
118 00 Prague, Czech Republic
WWW: http://nenya.ms.mff.cuni.cz
This thesis: http://nenya.ms.mff.cuni.cz/~mencl/phd-thesis/

Abstract:

A use case specifies only a part of the behavior of an entity. In general, use case modeling approaches do not address the issues of (i) assembling the behavior specified by several use cases, (ii) composing the behavior of communicating entities, nor (iii) reasoning on compatibility of the composed behavior.

In our work, we propose Generic UC View, a simple formal model capturing the key abstractions in use case modeling and the relations among them. We analyze the existing approaches (textual use cases, UML 1.5) and we propose Pro-cases, a specification mechanism based on Behavior Protocols [70] that supports behavior composition, behavior assembly and consistency reasoning. We describe how a Pro-case can created for a textual use case; we also describe how readily available linguistic tools can be employed to automate this conversion.

Subsequently, we analyze the emerging standard UML 2.0 and the four behavior specification mechanisms it provides, evaluating whether and how they address the above issues. Afterwards, we propose Port State Machine (PoSM), a behavior specification mechanism based on UML 2.0 Protocol State Machines, that fits into the UML 2.0 framework to model communication on a Port of a UML 2.0 structured classifier (component). Building on our experience with behavior protocols, we model an operation call as two atomic events request and response, permitting PoSM to capture the interleaving and nesting of operation calls on provided and required interfaces of the Port. Employing the semantics and formal tools available for behavior protocols, PoSMs support obtaining the assembled and composed behavior, as well as reasoning on behavior consistency within the framework of UML 2.0.

Keywords: behavior specifications, behavior assembly, behavior composition, UML, use cases

Preface

This thesis is based on results achieved over the course of several years of my doctoral study at the Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, and also within my stay at the Department of Computer Science of the University of New Hampshire.

The results have been documented in a number of papers; their bibliographical information is listed in Sect. 1.8. The key paper are published and/or accepted for publication; the most recent results are still work-in-progress and have not been published yet.

Where possible and suitable, the text from the respective paper is reprinted in the thesis in the exact form; we indicate this by using a sans-serif font. Where minor modifications were implemented to adjust the text with respect to whole context of the thesis, but the meaning of the paragraph has not been changed, we still use the alternative font, possibly making a record of the changes in a margin note. Text originally taken from one of these papers but substantially modified in this thesis is printed in normal font; we elaborate these conventions in Sect. 1.9.

Within a single chapter, the reprinted text in originates only from a single source, exceptions are chapters 1, 8 and 9. We denote the origin of the reprinted text with a margin note; the abbreviations used will be described in Sect. 1.9.

Acknowledgments

I would like to thank all those who supported me in my doctoral study and the work on my thesis. I very appreciate the help and counseling received from my advisor Prof. František Plášil. For the various help they provided me, I also thank my colleagues; a particular thank goes to Jiří Adámek, Tomáš Bureš and Petr Hnětynka.

For the help with understanding and employing the linguistic tools, a special thank goes to the members of the Institute of Formal and Applied Linguistics, in particular to Prof. Jan Hajič and Dr. Jan Cuřín.

My thanks also go to the institutions that provided financial support for my research work. In particular, the *Josef, Marie and Zdeňka Hlavkovi Foundation* and the *Bernard Bolzano Foundational Fund* have financially supported my participation at the OOPSLA 2001 conference. Through my doctoral study, my work was also partially supported by the Grant Agency of the Czech Republic projects 201/03/0911, 201/99/0244 and 102/03/0672 and the ITEA projects PEPiTA and OSMOSE.

I thank my mum and Ajka.

Table of Contents

Chapter 1

Introduction

1.1. Software Components and Behavior Specifications

The prevailing trend in software engineering is to design software systems by composition of *software components* [43]. Although there are many views on what constitutes a software component, the consensus is that a component is characterized by the services it provides and requires, typically via interfaces; the component models Darwin [47], Wright [3], SOFA [66] and Fractal [8] follow this paradigm. In addition, a component may be composed of nested (sub-)components; SOFA [66] and Fractal [8] are examples of such hierarchical component models.

The role of the component concept varies from unit of deployment [43] (expected to be compatible at the binary level, e.g., an *assembly* in dot-NET [19]) to design units which are subject to refinement in the evolution of a software system (e.g., SOFA [66], CCM [58]).

The external interface of a component is in general recognized as a first class entity; typically, it includes the list of services provided and required with their respective interfaces. To avoid conflict with the overloaded term *interface*, a different term is sought to refer to the external interface of a component. In the abstract model of Fractal, the term *membrane* is used to reflect the fact that it isolates the internals of the component from the environment; the SOFA component model [66] uses the term *frame*. In the CORBA Component Model (CCM) [58] and in the concrete model of Fractal [8], the definitions of the external interface of a component use only the term component, not emphasizing the difference between the component's external interface and the component itself.

The interfaces to services are described in terms of signatures of operations available on the interface. This allows to reason on correctness of component composition at the level of syntactical type correspondence. However, when considering composition of components obtained from different sources (vendors, development teams), such a specification is clearly not sufficient. A specification of the component's behavior is essential, describing the correct usage of the services provided by the component, the intended use of the services required by the component, as well as the dependence among interactions occurring on the individual interfaces of the component.

Often, behavior is specified only in an informal (plain language) description, in the form of additional documentation. Meanwhile, research in the field of formal methods has achieved a state where specification methods are available to specify the behavior of a component in a way that is (i) easy to learn, (ii) employs a human-readable notation, and (iii) provides means to reason on correctness of composition. An example of such a behavior specification method are Behavior Protocols [70] employed in the SOFA component model [66]. Besides textual notations, visual notations such as message sequence charts (MSC) and State-charts [28] and are used; here, verification methods employ generating test scenarios via simulation [16, 27], as well as translating the (complex) specification into a labeled transition system (LTS) as the common denominator and reasoning on the LTS [41].

Applying behavior specification methods to Component Based Software Development (CBSD) can yield significant benefits, either as the option to employ a tool to verify the

specification of the system being designed, or the possible CASE support in developing the component implementation. A CASE tool may utilize the behavior specification in deriving the initial design, as well as in generating skeleton code for operation implementations.

As use cases may be used as the starting point in specifying the behavior of a system or a component, employing use cases specifications to achieve these benefits would be very convenient. The chapters 1 and 2 of this thesis elaborate this issue and lay out the goals of the thesis.

1.2. Use Case Modeling

gettingWP

In principle a use case [33, 13, 60] is a description of a set of scenarios specifying how a set S of entities ought to communicate to achieve a certain goal; a communication is viewed as a sequence of events, such as a request or a response exchanged among the entities. Here, an entity can represent a system, a subsystem, an agent or a software component. Frequently, the use case is written from the perspective of one of those entities (*SuD*, system under discussion) – it specifies how SuD executes certain actions while communicating with other entities, *actors,* from S to achieve a specific goal. Basically, a *scenario* is considered a sequence of *actions* to be performed by SuD and the actors which reflects a particular case of their desired communication.

Use Case: MIS#1 Seller submits an offer
Scope: Marketplace
SuD: Marketplace Information System
Level: Primary Task
Primary Actor: Seller
Supporting Actor: Trade Commission
Main success scenario specification:
1. Seller submits information describing an item
2. System validates the description.
3. Seller adjusts/enters price and enters contact and billing information.
4. System validates the seller's contact information.
5. System verifies the seller's history to permit the seller to operate
6. System validates the whole offer with the Trade Commission
7. System lists the offer in published offers.
8. System responds with a uniquely identified authorization number.

Extensions:
2a Item not valid
    2a1 Use case aborted
5a Seller's history inappropriate
    5a1 Use case aborted
6a Trade commission rejects the offer
    6a1 Use case aborted

Sub-variations:
2b Price assessment available
    2b1 System provides the seller with a price assessment.

Figure 1.1 Sample Textual Use Case

Practitioners, e.g. [13, 40, 75] (also [64]), typically prefer use cases to be specified in plain English, to make them easily comprehendible to "wide audience". Such a use case is inherently informal, even though a predefined template is usually asked to follow. For example, such a template can be a form to be filled in to specify in a semi-programming way the desired set of scenarios (fig. 1.1).

There is a whole variety of ways different authors recommend to write use cases, ranging, e.g., from employing preconditions/postconditions in Catalysis [17], Use Case Maps [4], transition systems [78], to abstract state machines with the goal to generate test scenarios [27].

UML [60] includes a use case concept as well. It is, however, primarily focused on use case as an abstraction to capture the existence of a set of interaction scenarios among a set of actors and an SuD; it leaves the way internals of a use case are specified very open (the alternatives explicitly mentioned without any details include plain text, a state machine, activity graph, and specification via preconditions and postconditions). Rather, it concentrates on the relations among use cases (as a use case is a classifier in UML, a relation can be a dependency (extend/include) or generalization).

In general, the bottom line is that there are many different approaches and hard to compare techniques related to use cases, none of them being strongly recommended nor preferred; an overview is, e.g., in [31, 26].

Intuitively, there can be conflicts in use cases specifying two cooperating entities (separate SuDs). Even though there are many approaches to finding conflicts in dynamic and functional requirements, as pointed out in [29], they are frequently based on logic (and typically closely dependent on a particular use case technology) and require highly specialized experts to handle. This is in obvious contrast with practitioners' desire to make a use case easy to read and comprehend as mentioned above.

1.3. Textual Use Cases

While there are many approaches to writing use cases, we focus on the textual use case specification described in [13], which is based on a structured template (e.g., as the sample use case in Fig. 1.1); we choose this notation because of its broad recognition [12]. Here, the scenarios of a use case are determined by the *main success scenario specification* showing the "most typical" scenario, and by its *variations* and *extensions*.

natlang

The most typical scenario of a use case is specified in the main success scenario specification (top-level block), where each step describes an action that contributes to achieving the goal of the use case; steps in a block are labeled with a step number. Alternative successful flows are specified in the sub-variations section; exceptional situations (error handling) are specified as extensions. Both variations and extensions are attached to a step (by its label), start with a condition (guard), followed by a sequence of steps forming a nested block; here, step labels are prefixed by the extension/variation label. A nested block typically ends with a special action – either moving the flow to a step within the enclosing block, or causing the enclosing block to end. (In case no explicit special action is used, the flow implicitly resumes with the next step of the enclosing block). With special actions, extensions can be used to model repetition or skip a part of the enclosing block. The use case in Fig. 1.1 demonstrates most of the features described here.

Use Case: MIS#3 Buyer buys a selected item
Scope: Marketplace
SuD: Marketplace Information System (MIS)
Level: Primary Task
Primary Actor: **Buyer**
Supporting Actor: **Seller, Credit Verification Agency**
Main success scenario specification:
1. Buyer chooses to accept a selected offer.
2. System validates the offer.
3. User enters billing information, select a payment method and provides the payment details.
4. System validates the buyer's information with the Credit Verification Agency.
5. System performs the sale.
6. System informs the seller that the offer has been accepted and provides the shipping information.
7. System transfers the payment to the sellers account.
8. System responds to the buyer with an uniquely identified authorization number.

Extensions:
2a Offer is not valid
2a1 Use case aborts.
Sub-variations:

Use Case: TC#1 Validate an offer
Scope: Marketplace
SuD: Trade Commission (TC)
Level: Primary Task
Actors: **Marketplace Information System**
Main success scenario specification:
1. Marketplace Information System requests the Trade Commission to validate an item.
2. Trade Commission determines the item category.
3. Trade Commission validates the item with the restrictions imposed by the item's category.
4. Trade Commission confirms the item is valid.

Extensions:
3a Item breaches the category restrictions
3a1 Trade Commission notifies the Marketplace Information System that the item is not valid.
3a2 Use case aborts.
Sub-variations:

Figure 1.2: Additional textual use cases

1.4. Assembly and Composition

We illustrate the issues of assembly and composition on a sample marketplace application. The scope diagram in Fig. 1.3 shows all the entities involved in this example, their nesting and communication links. Figure 1.1 shows a sample textual use case of the entity Marketplace Information System (MIS); Fig. 1.2 shows an additional use case of the entity MIS and a use case of the Trade Commission (TC); for simplicity, we use the plain English textual notation. The use case MIS#1 demonstrates the use of extensions to handle failures and variations to specify alternative flows; the use cases in Fig. 1.2 employ extensions only. In a use case header, *scope* means the parent entity; the other elements have obvious meaning.

uml20uc (work in progress); with minor modifications (changed example)

Apparently, in addition to "Seller submits an offer" and "Buyer buys a selected item", there would be more use cases of MIS – all of them together specifying the behavior of MIS. The scenarios specified by these individual use cases may be required to be performed in a sequence, concurrently, etc.; in general we call the

process of “putting use cases together” *behavior assembly* which yields the *assembled behavior* of a SuD. In this view, the *use case model* of MIS is determined by all its use cases and by the way they are combined.

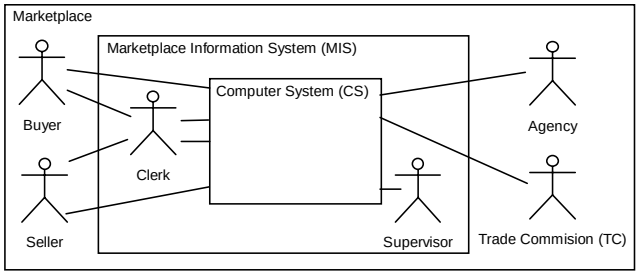


Figure 1.3: Scope diagram of the Marketplace system

In a similar vein, for composed entities the question is whether it is possible to obtain the behavior of the containing entity by *composition* of the assembled behaviors of the nested entities. The *composed behavior* has to capture the internal communication among the nested entities, as well as those of their activities that are visible outside as the behavior of the containing entity. In the example above, MIS communicates with TC in such a way that the steps 1 and 4 of TC#1 correspond to the step 6 of MIS#1. Except for this special case (synchronization), in which the complementary communication actions become an internal activity, TC and MIS operate in parallel and their activities interleave. The communication of MIS and the Seller reaching outside of MIS gives an example of an externally visible communication.

The semantics of textual use cases is typically defined only intuitively. Usually, it is possible to figure out what is the meaning of simple operations for behavior assembly (alternative, sequencing, repetition); however, there is typically no precise semantics of concurrent execution. Moreover, as communication links and complementary communication actions are not explicitly captured, it is not possible to define semantics for behavior composition.

Importantly, if composition semantics was precisely defined for a specification mechanism, it might be possible to reason on behavior consistency of nested entities, e.g., whether the composed behavior of Clerk, Computer System and Supervisor is consistent with the assembled behavior specified for MIS.

1.5. Problem Statement

Typically, use cases are written only informally, as plain language descriptions. In CBSD, employing formal methods in use case specifications might provide beneficial results. Capturing the “whole picture” of the behavior of an entity (a component) may be used by a tool to aid in creating the design of the component’s implementation. Moreover, when supported by the formalism employed, behavior reasoning may be used to validate the composition of components in the system under development.

When use cases are used to capture the requirements of a future system, the behavior specification is scattered across the set of use cases. Although several authors propose formal methods for specifying use case behavior, the issues of obtaining the “whole picture”

of an entity's behavior, behavior assembly and behavior composition have not been explicitly considered for use cases so far, neglecting the possible benefits. When permitted by the formalism employed, both the design and validation features might be implemented in a tool, possibly as a plugin to a CASE (UML) environment.

1.6. Goal of the Thesis

The goals of the thesis are to explore the relations among use cases employed to specify behavior of a software system. The initial goal (i) is to analyze and formally define the concepts of behavior assembly and behavior composition. Here, a supplementary goal (ii) is to develop a formal model to capture these concepts; this model should also serve as the basis for consistency reasoning. Further, goal (iii) is to employ this formal model in analyzing traditional use case approaches to see how the concepts of behavior assembly and composition are reflected. The analysis should cover at least the textual use cases [13] and the support of UML 1.4/1.5 [59, 60] for use cases. A subsequent goal (iv) is to utilize the analysis results and propose a use case technology reflecting behavior assembly and behavior composition with support for reasoning on behavioral compliance; a related goal (v) is to explore the options to automate the conversion of textual use cases into this proposed technology.

Afterwards, the goal (vi) is to focus on the upcoming standard UML 2.0 and analyze the behavior specification mechanisms it provides. The adjunct goal (vii) is to propose an extension of UML 2.0 with a specification mechanism providing support for behavior assembly, behavior composition and behavioral reasoning.

1.7. Structure of the Thesis

In Chapter 2, we elaborate our view on use cases and propose *Generic UC View*, a generic model to interpret use cases, addressing goals (i) and (ii). Based on this model, the issues to consider in consistency reasoning are articulated. Next, in Chapter 3, we analyze the traditional textual use cases and the view of UML 1.4/1.5 on use cases, addressing the goal (iii). Chapter 4 addresses the goal (iv) by proposing Pro-cases, a specification mechanism that addresses the issues articulated in Chapter 2; we also outline conversion of textual use cases into Pro-cases, partially addressing goal (v). In Chapter 5, we explore the options of employing readily available linguistic tools to automate the conversion of use cases to Pro-cases, completing the answer the goal (v). In Chapter 6, we address the goal (vi) by analyzing the support provided by UML 2.0 for use cases, and the behavior specification mechanisms considered for specifying use case behavior. In Chapter 7, we propose Port State Machines, a UML specification mechanism applying the idea of Pro-case, giving an answer to the goal (vii).

1.8. Contribution and Publications

The Generic UC View was originally published in the proceedings of the IDPT 2003 conference [67]; at the conference, the paper received the *Rudolf Christian Carl Diesel Best Paper Award*. Subsequently, a slightly modified version of the paper was published in the Transactions of the SDPS: Journal of Integrated Design and Process Science [69]. The contribution of the two authors of these papers is equal. Further, Generic UC View is also

described in TR 02/11 of the Department of Computer Science, University of New Hampshire; the appendices of the technical report feature an elaborate example of use case models.

The recent work on transformation of textual use cases into Pro-cases, described in Chapter 5, is still work-in-progress; it has been documented in [52] but has not yet been published.

The same applies to the recent analysis of UML 2.0 behavior specification mechanism, the key topic of Chapter 6. The analysis is still work-in-progress; it has documented in [51] but has not yet been published. This work is a joint effort; the contribution of the three authors of this paper is equal.

The Port State Machines, proposed in Chapter 7, have been presented at the workshop *Compositional Verification of UML Models*, held as a part of the UML 2003 conference. An extended version of the contribution [49] has been accepted for publication in a special volume of the Electronic Notes in Theoretical Computer Science, published by Elsevier Science. Port State Machines are also described in Tech. Report No. 2003/4 of the Department of Software Engineering, Charles University in Prague [48].

Results from the earlier work on component based software development (not included in the thesis) have been captured in a poster presented at the OOPSLA 2001 conference [50] (extended abstract published in the OOPSLA 2001 Conference Companion) and have been documented in a number of technical reports [54, 53, 55].

Reviewed articles

[67] Plasil, F., Mencl, V.: *Getting "Whole Picture" Behavior in a Use Case Model*, in Proceedings of IDPT 2003, Austin, Texas, U.S.A., Dec 2003, ISSN 1090-9389

[69] Plasil, F., Mencl, V.: *Getting "Whole Picture" Behavior in a Use Case Model*, in Transactions of the SDPS: Journal of Integrated Design and Process Science, vol. 7, no. 4, pp. 63-79, Dec 2003, ISSN-1092-0617

[49] Mencl, V.: *Specifying Component Behavior with Port State Machines*, Accepted for publication in proceedings of Compositional Verification of UML Models workshop (Oct 21, 2003, part of UML 2003) in a volume of the Electronic Notes in Theoretical Computer Science, Elsevier Science

[50] Mencl, V.: *Autonomous Points in Component Composition*, Extended abstract of the Poster presented at OOPSLA 2001, in the Conference Companion, ACM ISBN: 1-58113-441-X, Tampa, FL, USA, Oct 2001

Work in progress to be submitted

[51] Mencl, V., Plasil, F., Adamek, J.: *Use Cases in UML 2.0: Analyzing Support for Behavior Assembly and Composition*, work-in-progress

[52] Mencl, V.: *Converting Textual Use Cases into Behavior Specifications*, work-in-progress

Technical reports

[48] Mencl, V.: *Enhancing Component Behavior Specifications with Port State Machines*, Tech. Report No. 2003/4, Dep. of SW Engineering, Charles University, Prague, Sep 2003

[68] Plasil, F., Mencl, V.: *Use Cases: Assembling "Whole Picture Behavior"*, TR 02/11, Dept. of Computer Science, University of New Hampshire, Durham, 2002

[53] Mencl, V., Adamek, J., Buble, A., Hnetynka, P., Visnovsky, S.: *Enhancing EJB Component Model*, Tech. Report No. 2001/7, Dep. of SW Engineering, Charles University, Prague, Dec 2001

[54] Mencl, V.: *Managing Configuration of Update-enabled Software Components*, Tech. Report No. 2001/5, Dep. of SW Engineering, Charles University, Prague, Oct 2001

[55] Mencl, V., Hnetynka, P.: *Managing Evolution of Component Specifications using a Federation of Repositories*, Tech. Report No. 2001/2, Dep. of SW Engineering, Charles University, Prague, Jun 2001

Presentations

Besides conference presentations, the results have been presented in talks given at cooperating universities. Below is the list of presentations (given by the author of the thesis, unless noted otherwise).

Mencl, V.: *Specifying Component Behavior with Port State Machines*, presented at Colloquium CIS-TU Berlin and Fraunhofer-ISST, Berlin, Apr 2004

Mencl, V.: *From Textual Use Cases to Behavior Specifications*, presented at Colloquium CIS-TU Berlin and Fraunhofer-ISST, Berlin, Apr 2004

Plasil, F., Mencl, V.: *Getting "Whole Picture" Behavior in a Use Case Model*, presented at CS900 Colloquium seminar, Dept. of CS, University of New Hampshire, Durham, NH, U.S.A., Dec 2003

Plasil, F., Mencl, V.: *Getting "Whole Picture" Behavior from Use Cases*, presented by F. Plasil at Informatics Colloquium, Masaryk University, Brno, Apr 2003

1.9. Note on Conventions Used

This thesis is based on several papers published, accepted for publication, or being work-in-progress, as declared in the previous section. With the intention to preserve text already finalized for publication as much as possible, where possible and suitable such text is reprinted in the thesis in the exact form.

We distinguish text copied verbatim by using a Sans-Serif font.

Where it was necessary to modify the original text in a way not changing its meaning, the annotation inserted is distinguished by using the regular Serif font (Times New Roman); margin notes indicate modifications to the original text and the nature of the modifications.

The source of text copied verbatim is denoted with margin notes; we use the following abbreviations:

For published work:

`gettingWP` for *Getting "Whole Picture" Behavior in a Use Case Model* [67], also published in [69], used in Chapters 2, 3 and 4

`posm` for *Specifying Component Behavior with Port State Machines* [49], forming Chapter 7

For work in progress:

`natlang` for [52] *Converting Textual Use Cases into Behavior Specifications*, forming Chapter 5

`uml20uc` for *Use Cases in UML 2.0: Analyzing Support for Behavior Assembly and Composition* [51], used in Chapter 6.

Chapters 1, 8 and 9 employ text from several sources, we denote the origin of each section and/or paragraph with a margin note as appropriate.

Note that different on-the-fly examples have been used in the papers this thesis builds upon. Consequently, different examples are used throughout this thesis – these examples are unrelated, but demonstrating the our ideas in a similar way; a goal considered when while writing the thesis was to preserve the text in the form it was published.

While analyzing UML 2.0, we developed extensions to the formal model Generic UC View originally published in [67, 69]. For consistency with already published results, we have decided to describe Generic UC View in Chapter 2 in its original form; the extensions developed for analyzing UML 2.0 will be presented in Chapter 6, devoted to UML 2.0 analysis [51].

Chapter 2

Use Cases Elaborated

gettingWP

2.1. Generic UC View – Motivations and Goals

In [70, 71], our research group developed an agent model, where the agents process sequences (traces) of atomic *events*, and introduced a way to describe (approximate) the agents' behavior via behavior protocols, which was applied on software components in SOFA in order to specify component behavior and test behavior compliance of components, including neighboring levels of nesting. Based on this experience, we realized that similar compliance checks should be done also for use case models associated with component-based (interface-centered [17]) design.

Here, as a variety of different use case techniques might be considered, the key question is what the basic relations among the behaviors of entities in use cases are, provided these relations should allow for capturing the behavior relationships among the proposed components in a hierarchical system. For instance, whether the composed behavior of components (at a particular level of nesting), specified separately for each of the components, corresponds to the behavior specified for the parent component, etc. Another question is what the basic relations upon a set of scenarios are, in order to define some "reasonable" behavior concepts and relations among them (associated with use case entities) which could be easily interpreted in classical formal tools, such as state machine, labeled transition systems, etc. Based on theses objectives, the chapters 2, 3 and 4 (originally, the papers [67, 69]) aim at these goals:

(1) Finding a generic view on use cases in order to articulate key abstractions allowing to capture behavior compliance of entities/actors at different levels of their decomposition, resp. refinement, and identifying which relations should be chosen as the basis for such reasoning.

(2) Showing how the abstractions in the proposed generic view correspond to the use case related concepts in UML and how the classical textual, system centric use case specifications can be mapped/interpreted in terms of these abstractions.

(3) Introducing a use case technique which would feature these abstractions and thus provide reasoning, while still simple enough to be easy to apply in practice (emphasis on readability and easy comprehension); for instance, transforming a textual use case to the form the new use case technique would require should be an easy step.

Reflecting this aim, the relevant chapters of this thesis are is organized as follows: Sect. 2.2 of this chapter targets the goals (1) by providing Generic UC View, Chapter 3 addresses goal (2) by providing comparison of Generic UC View to textual use cases and UML, while Chapter 4 addressed the goal (3) by introducing Pro-cases (short for Protocol Use Cases).

2.2. Generic UC View

To provide a basis for reasoning about the key abstractions (and capture their relationship) in the traditional use case modeling [13, 33, 35, 60], we introduce the following generic model (*Generic UC View*).

Basic concepts. Assume an entity S is composed of sub-entities A_1, ... ,A_n. By definition S forms the *scope* of A_i; the topmost scope is called *system*. An entity A_i communicates through communication links (*connections* for short) with (1) other (*actors*) A_j of the scope S, and potentially (2) with other external actors located in the parent scope, i.e., in the scope of S. In case (1), the communication is observed on the *internal* connections of S, while in case (2) on *external* connection of S. Advantageously, the nesting of entities and their scopes can be expressed as a scope diagram. Here, by convention, the stick-figure symbol denotes an entity which is an abstraction of a particular human role, while a rectangle denotes an entity which activity is at least partially software driven (typically a SuD), and a line represents a connection. For example, in Fig. 2.1(a), A_1, A_2, and A_3 are in the scope of $\overline{A}_2$. Here, C_1 and C_3 are external connections of $\overline{A}_2$, while C_2 and C_4 are internal connections of $\overline{A}_2$. In a similar vein, C_1, C_2 and C_3 are the external connection of A2. Furthermore, $\overline{A}_1$, $\overline{A}_2$, and $\overline{A}_3$ are in the scope S. Frequently, not all of the levels of entity nesting are captured on a scope diagram, typically leaving out the targets of the external connections of the outmost scope (fig. 2.1(b)).

Note that a scope diagram captures the relations among entities, not among use cases and entities (as UML use case diagram does).

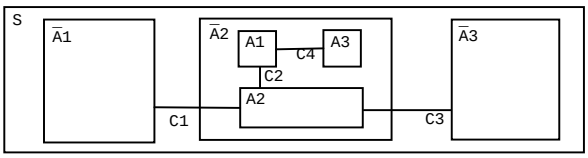


Figure 2.1 (a): Scope diagram of entity S

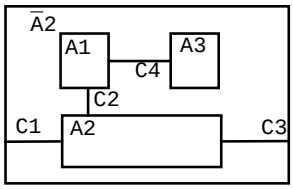


Figure 2.1 (b): Scope diagram of $\overline{A}_2$

Example 2.1 Figure 1.3 used in the previous chapter shows the scope diagram of the sample Marketplace application. Within the scope of Marketplace (a business entity), the actors Buyer, Seller, Agency, Trade Commission and the Marketplace Information System and their connections are visible. Inside the Marketplace Information System are three entities: the Clerk, the Supervisor and the Computer System. The use case demonstrated in Fig. 1.1 describes the interaction of the Marketplace Information System with its surrounding actors in the scope of Marketplace.

Scenarios. A particular way of communication of an entity A on its connections in a run of system Σ is captured as a *scenario* s ∈ Scenarios. All the scenarios of A in any run of Σ form the *behavior* Com(A) ⊆ Scenarios. On the domain Scenarios we assume the existence of a subscenario relation (partial order).

By convention, Com(A)/ExConn(A) is the behavior of A restricted to its external connections (while Com(A)/InConn(A) is restricted to the internal connections of A); here, Com(A)/Conn denotes restriction of scenarios from Com(A) to the communication observed on the connections from a set of connections Conn.

Assuming an entity C is composed of entities A and B, Com(C) is composed of Com(A) and Com(B), written Com(A) $\sqcap_X$ Com(B), where X = ExConn(A) ∩ ExConn(B), in such a way that

(1) together with Com(A) / InConn(A) ∪ Com(B) / InConn(B) the behavior on the joint connections between A and B becomes Com(C) / InConn(C),

(2) (Com(A) / ExConn(A) ∪ Com(B) / ExConn(B)) –
(Com(A) / ExConn(A) ∩ Com(B) / ExConn(B))

becomes Com(C)/ExConn(C) (where – stands for set subtraction)

The *subscenario* relation permits us to further constrain the possible interpretations of composition. The condition (1) requires that communication on the internal connections of A and B, together with communication on the connections between A and B, becomes communication on the internal connections of C; by employing the *subscenario* relation; we require that the following conditions hold:

∀ s ∈ Com(A) / InConn(A) : ∃ s' ∈ Com(C) / InConn(C) : s subscenario s'
∀ s ∈ Com(B) / InConn(B) : ∃ s' ∈ Com(C) / InConn(C) : s subscenario s'
∀ s ∈ Com(A) / X : ∃ s' ∈ Com(C) / InConn(C) : s subscenario s'
∀ s ∈ Com(B) / X : ∃ s' ∈ Com(C) / InConn(C) : s subscenario s'

In a similar vein, we constrain the condition (2), requiring that the communication on the external connections of A and B becomes communication on the external connections of C, by requiring that:

∀ s ∈ Com(A) / (ExConn(A) - X): ∃ s' ∈ Com(C) / ExConn(C): s subscenario s'
∀ s ∈ Com(B) / (ExConn(B) - X): ∃ s' ∈ Com(C) / ExConn(C) : s subscenario s'

Use case model. Let U^A be the set of *basic use cases* (behavior specifications) where A is the SuD. A use case $UC^A_i \in U^A$ describes/generates a set of scenarios; so, by convention, we write also Com(UC^A_i) ⊆ Scenarios. Further, we define a binary relation includes such that UC^A_j includes UC^A_k means that the specification of UC^A_j includes (refers to) the specification of UC^A_k (macro substitution idea, thus no circular dependencies allowed). Also we assume that if $UC^A_j \in U^A$ and UC^A_j includes UC^A_k then also $UC^A_k \in U^A$; moreover we require UC^A_j includes UC^A_k to imply that for any s ∈ Com(UC^A_k) there exists an s' ∈ Com(UC^A_j) such that s subscenario s' (*subscenario preservation*).

Notice that even if Com(UC^A_k) = Com(UC^A_j), not necessarily $UC^A_k = UC^A_j$; thus different specifications can generate the same behavior. Also, in the need to distinguish the elements of U^A we do so by subscripts, writing e.g. UC^A_k (this also reflect that use cases of A are enumerated in a typical use case technology).

A *use case model* of A (denoted UM^A) is a set of *use case expressions*, *(*also *use cases* for short), where an expression UE^A (syntactically in principle) generates a set of scenarios (by convention we write Com(UE^A) ⊆ Scenarios). A use case expression UE^A is either a basic use case UC^A_i or is composed by applying

operations from a set of operations UEop on their operands – (sub)expressions, assuming some priorities, parenthesis, etc. apply. The semantic of these operations can include sequencing, parallel composition of the scenarios generated by the operands, etc. (as illustrated in Sect. 4.2). The includes relation can be naturally extended to use case expressions.

Whole picture behavior. As a use case $UE^A_j \in UM^A$ provides only a partial "j-th" description of A's behavior ("the whole picture behavior" of A),the *assembled behavior* of UM^A is defined as $Com(UM^A) = \cup_j Com(UE^A_j)$, $UE^A_j \in UM^A$. To emphasize an important special case, we say that UM^A *has a representative* if there is UE^A in UM^A such that $Com(UM^A) = Com(UE^A)$; (also: UE^A is an *representative* of UM^A).

2.3. Consistency Issues

Inherent to refinement/synthesis steps in a design of a specified system Σ is the necessity to combine behavior specifications in order to get "the whole picture" behavior specification and capture the behavior specification compliance of several cooperating/nested entities. In particular, the following four issues are closely related to this necessity:

(a) Does the combined behavior, as specified by all UM^{Ai} in a scope S, comply with the behavior specified for S in UM^S.

(b) Does the assembled behavior, as specified by a UM^A, really reflect the desired behavior of A (as this is hard to address directly, we will consider equivalence checking – decidability whether two use case models UM^A and 'UM^A specify the same behavior, i.e., $Com(UM^A) = Com('UM^A)$).

(c) Is the desired communication between A_i and A_j via their connection(s) in S really reflected in the behavior as specified (separately) by UM^{Ai} and UM^{Aj}.

(d) If there is no representative of UM^A and the behaviors of use cases in UM^A overlap, is there a way to find/construct a representative in order to get a "whole picture behavior" directly from the specifications, without the need to generate scenarios.

In general, addressing (a) and (b) requires defining a *behavior compliance* as a binary relation upon the behavior of entities. Intuitively, Com(A) compliant with Com(B) if B can be replaced in Σ by A by taking over all its external connections in such a way that "B behaves in A's place as it were A". The issue (c) can be addressed by finding a binary relation consent upon behavior of entities: Intuitively, Com(A) consent Com(B) if there is no inconsistency resp. "erroneous scenario" in the behavior of A and B on their joint connections. To define these relations (and $\sqcap_X$) precisely, a specific interpretation of Scenarios and the relations/operations available for it have to be known.

Note that the relations are (intentionally) defined only for Com(A) and Com(B) and not A and B, as in a concrete UC view, behavior of A will be approximated by $Com(UM^A)$. Sect. ? will define such an approximation for the Pro-case model; for more information, please refer to [70].

Assuming the existence of compliant with and consent, the issues (a) - (c) can be rephrased as

(a) $Com(UM^{A1}) \sqcap_{X1} Com(UM^{A2}) \sqcap_{X2} .... \sqcap_{Xn-1} Com(UM^{An})$ compliant with $Com(UM^S)$, (here X_i is $ExConn(A_i) \cap ExConn(A_{i+1})$) and

(b) Com(UM^A) compliant with Com('UM^A) and Com('UM^A) compliant with Com(UM^A)
(c) Com(A_i) consent Com(A_j)

Obviously a big advantage can be taken of extending the definitions of compliant with, consent and $\sqcap_X$ to UM^A (to make them applicable not only on Com(A), i.e. the behavior itself but also on the behavior specification). Then, assuming a set UEop is "reasonably" defined, the consistency issues (a) - (d) can be addressed by reasoning on uses case expression (for an example see Chapter 4).

An example of defining the compliant with and consent relations is available in [1, 70]. The compliance relation is defined in Sect. 4.1 and also in Sect. 7.3.1 of this thesis; both these sections also describe and discuss the consent operator.

2.4. Goals of the Thesis Elaborated

In Sect. 1.6, we briefly outlined the goals of this thesis. The goal (i) is to explore the relations among use cases, with respect for employing the use cases to specify the behavior of a hierarchical software system at various levels of nesting; the supplementary goal (ii) is to develop a formal model to capture these concepts. By presenting Generic UC View (GUCV) in this chapter, we have already addresses the goals (i) and (ii). By proposing GUCV, we have identified the key abstractions in use case modeling and relations between them. Using GUCV, the concepts of behavior assembly and behavior composition are defined; moreover, GUCV also articulates the three issues in consistency reasoning.

The goal (iii) is to use the formal tools provided by GUCV to analyze how the concepts identified in GUCV are reflected in traditional use case approaches. The analysis should cover at least the textual use cases [13] and the support of UML 1.4/1.5 [59, 60] for use cases. The analysis should examine:

1. how are the basic abstractions interpreted (entity, use case)
2. whether and how the assembled behavior (representing "whole picture" of the behavior) of an entity can be obtained.
3. how the consistency issues are addressed.

The goal (iv) is to utilize the results of these analyses and propose a use case technology that provides an interpretation for the abstractions defined by Generic UC View; the proposed technology should support behavior assembly, behavior composition and consistency reasoning; in consistency reasoning, the relations addressing consistency reasoning should be decidable in a computationally feasible way. A related goal (v) is to propose conversion from the textual use cases to this notation; to prove the feasibility of the proposal, another part of this goal is to implement a tool transforming textual use cases into this formal notation.

Next, the goal (vi) is to analyze the behavior specification mechanisms available in UML 2.0, providing four concrete behavior specification mechanisms in addition to the generic framework for capturing relations among use cases. Here, in order to handle the diversity of the specification mechanisms available in UML 2.0, an additional task is to extend Generic UC View in a way that permits analyzing these specification mechanisms defined at varying levels of clarity and unambiguity.

The analysis to be performed includes examining:

1. how are the basic abstractions interpreted (entity, use case)

2. how is the set Scenarios interpreted (and whether its internal structure is defined); this includes how the behavior of a use case (the set of scenarios) is defined
3. whether obtaining the whole picture behavior can be obtained and what the operations available for assembling behavior are
4. what is the support for representing the whole picture behavior with a single representative use case; this includes analyzing whether the operations for assembling behavior discovered can be interpreted via some native operations in the given specification mechanism.
5. whether the composed behavior can be obtained
6. how the consistency reasoning is addressed.

Consequently, the goal (vii) is to employ the analysis results and propose a UML specification mechanism (proposed as an extension to UML 2.0) that supports consistency reasoning, as well as behavior composition, behavior assembly, and interpreting the assembly operations via native operations of the specification mechanism.

Chapter 3

Analyzing Textual Use Cases and UML 1.5

gettingWP

3.1. Textual Use Case Analysis

There is a variety of use case techniques/technologies based on textual specification of UC^A_i (ranging from unstructured narratives to semi-structured scripts, see overviews in [31, 26]). In this section we analyze the textual use case specification described in [13], which is based on a structured template (fig. 1.1). We choose it because of its broad recognition [12].

Basic concepts. The central concept, actor, has an important special case: stakeholder, having a goal. Scope is introduced as the entity A for which a UC^A is written (A is SuD). Communication links (connections) are introduced indirectly by identifying the cooperating actors in each UC^A.

Scenarios. A scenario is "sequence of steps showing how actions and interactions unfold" (thus sequence of interacting actions in principle); however, the domain Scenarios is not explicitly defined. The subscenario relation is not introduced explicitly, being considered only indirectly, as a consequence of the includes relation (below).

Use cases and relations. A goal *k* of a stakeholder H with respect to an entity A (considered as SuD and also scope) is identified as the reason for writing a use case UC^A_k. To emphasize that UC^A_k is focused on a goal of A, the stakeholder H is also called (ultimate) primary actor in UC^A_k. The description of communication of A with external entities is "white-box" based, i.e. all external actors of A are visible, regardless of how deep is A nested in the entity hierarchy forming Σ. On the contrary, it is strongly discouraged to specify any communication of A with internal entities in any UC^A_k (black box view recommended). No UM^A is introduced, but there is the concept of Σ (SuD, the computer system to be designed).

As illustrated in Sects. 1.3 and 1.4, the structure of the textual descriptions is defined as a template, providing generic guidelines as to how to specify the generated scenarios by means of main success scenario, its extensions, and variants, plus which fields should be included in the header of a use case (Stakeholder, Scope = SuD, etc.).

Addressing includes in principle, a sub use case relation is defined with the semantics UC^A_j "calls" UC^A_k where acyclic property is silently assumed and indirectly supported by introducing specific levels of nested use case calls (cloud, sea-level, underwater). Thus, the subscenario preservation is silently guarantied on the domain assumed for UC^A_j. Here, any top-most use case is also a *summary use case* – this roughly corresponds to the choice of use cases to form a UM^A. Moreover, inspired by UML, the extends relation is defined, based on the idea of ex post defined extension points in a use case: UC^A_j extends UC^A_k means that UC^A_j explicitly states which parts of UC^A_k are the extension points (could be described by a set of context rewriting rules). Generalization (again UML inspired) is not explicitly considered in the recommended template (using generalization is discouraged).

For UC^A_j, $Com(UC^A_j)$ is explicitly defined as the set of scenarios generated by UC^A_j. Com(A) is considered the collection of scenarios where A is involved as SuD (generated by a UC^A_j) – considering just summary use cases to be involved in behavior assembling is silently assumed.

Whole picture behavior. The idea of selecting use cases from U^A to form UM^A is reflected in the concept summary use case. In principle, the summary use case idea implies nesting and sequencing of use cases from U^A (via includes or extends) in the form described by a higher level use case from UM^A. On the other hand, the precondition and postconditions of a UC^A_i and UC^A_j from UM^A allow for partial ordering of the behaviors $Com(UC^A_i)$ and $Com(UC^A_j)$, including a parallel composition; here, preconditions can be interpreted as unary operations from UEop for which only basic use cases are allowed as operands. As the semantical spectrum of behavior composition via preconditions can be really broad, the original paper on use cases [34] assumes that no concurrency is modeled in a use case.

Addressing consistency issues. No compliant with, consent, nor $\sqcap$ are defined, therefore none of the key consistency issues is addressed (except for reasoning "by hand" on an intuitive basis).

3.2. UML 1.4/1.5 Analysis

Basic concepts. The UML use case package (a subpackage of the Behavior Elements package) is a very generic framework for specifying use cases. UML specifies the actor and use case as the central concepts. These corresponds to our entity A and UC^A concepts. The idea of communication links (connections) in Generic UC View is reflected in UML very indirectly: the information about connections among the entities in a subsystem can be obtained only by a systematic walk-through of the use cases (for these entities) and recording their associations with actors – an use case of an entity (SuD) A contains (as an association) the info with which external entities (actors) A communicates. As to nesting of entities, a UML actor can model (not be) a system, subsystem, or class; in a hierarchical system, it is very hard to imagine capturing the communication among entities (their behavior) at such different abstraction levels under the circumstances that the info on their communication is so hard to get.

Scenarios. A scenario is a *use case instance* (no Scenarios introduced). As to the way of how a UC^A is actually specified, UML is very generic (p.2-142): *"A use case can be described in plain text, using operations and methods together with attributes, in activity graphs, by a state machine, or by other behavior description techniques, such as preconditions and postconditions."* Even though UML considers nested entities, it does not bring an explicit scope concept.

Use cases and relations. UML is specific on the relations defined for U^A. It defines the include, extend, and generalize relations, but in a very generic way: As to include, it requires nothing more than *"a use case contains the behavior defined in another use case"*. We interpret this in such a way that include corresponds to our includes (and also that subscenario preservation is required by the UML include).

The extend relation and generalize relation are defined for use cases UC^A_i and UC^A_j, but in a very general way. While generalization allows for systematic modifications of scenarios (one could imagine employing rewriting rules for this purpose), extends allows only for insertion of several subscenarios into an existing scenario at its predefined extension points. But elsewhere it is argued that these

relations are not well defined, e.g. [13, 26, 22]. In addition, there is the concept of superordinate and subordinate use case (below).

Whole picture behavior. There is no concept similar to Com(UM^A), so it is not clear where "the generation of scenarios really starts". Using the terminology from Sect. 2.2, consider UC^A_i and UC^A_j from a U^A and assume UC^A_i include UC^A_j and $UC^A_i \in UM^A$; it is not clear whether $UC^A_j \in UM^A$, i.e. whether UC^A_j contributes also to Com(UM^A) or produces just subscenarios employed in Com(UC^A_i).

Consistency issues. The issues (a), (b) are addressed in only in a very general way by asking (p. 2-145): *"Thus, if use cases are used for the specification part of the system package, these use cases are equivalent to those in the use-case model of the system; that is, they express the same behavior but possibly slightly differently structured."* and *"Furthermore, if several models are used for modeling the realization of a system (for example, an analysis model and a design model), the set of use cases of all system packages and the use cases of the use-case model must be equivalent."*

In addition (a) is also, again in a very general way, addressed by the subordinate/superordinate relations upon UC^{Ai}_j and UC^S where A_i are in the scope S by asking all UC^{Ai}_j subordinate to UC^S to "cooperate" in a way described by a collaboration diagram. No compliant with, consent, nor $\sqcap$ are defined, therefore none of the key consistency issues is addressed. Also, no use case expressions (and thus no operations UEop) are considered.

3.3. Analysis Summary

In this chapter, we have analyzed the textual use cases and the support for use cases in UML 1.4/1.5. The basic concepts (entity, SuD, actor, connection) are reflected in both of these technologies. However, none of them explicitly defines the domain *Scenarios* (although for textual use cases, *Scenarios* is implicitly assumed). Consequently, composition is not defined for UML 1.x and is difficult to interpret for textual use cases.

No operations for assembling behavior are defined for UML 1.x. For textual use cases, sequencing, repetition and alternative can be used; semantics of parallel composition cannot be reasonably defined. The traditionally used summary use case addresses only the simple case of exclusively using the sequencing operation.

Both the technologies have the notion of a use case model, however, the information for constructing the whole picture behavior is not captured in UML, and only in a limited form in textual use cases.

With no concrete specification mechanism prescribed, the issue (d) of constructing a representative use case is not addressed in UML 1.x. In textual use case, a use case may be constructed for use case expressions employing only the operations sequencing, repetition and alternative, however, for complex expressions, the resulting use case would be very hard to manage; in the traditional textual use case recommendations, nested extensions are discouraged.

As no scenarios domain is provided (and as the composition operation cannot be interpreted), consistency reasoning is not addressed in neither of these technologies.
Please note that the analysis in Sect. 3.2 applies only to UML 1.4/1.5 [59, 60]; UML 2.0 [61] will be analyzed in Chapter 6.

Chapter 4

Pro-cases

gettingWP

In the previous chapter, and in particular in Sect. 2.3, we saw what the benefits of the relations compliant with, consent, and the operations for creating use case expressions could be, if defined for a particular specification mechanism. In this chapter, we present a concrete instance of our Generic UC View – *Pro-cases* (short for *Protocol use cases*), a specification mechanism based on Behavior Protocols [70, 71], developed within our group. Pro-cases features all the relations asked for in Sect. 2.3, providing a proof of the Generic UC View concept; in addition, the compliance relation is decidable and a verifier tool is available [46, 77].

4.1. Behavior Protocols Overview

Behavior Protocols [70, 71] employ an event-based model of communication of nested agents. Behavior is modeled in terms of events processed by an agent (a computational entity); an event may be either emitted (denoted *!a*), absorbed (*?a*) or internally processed (τa). An agent may be either *primitive* or *composed* (of other agents). Agents communicate via a set of *connections*. A particular run of an agent A is captured as a finite sequence of atomic events (*trace*) processed by A; the events are from a finite domain denoted ACT. Behavior of A (denoted Com(A)) is captured as the set of all traces of A, forming a language upon ACT. In [70], L(A) is used for the language; to be consistent with the Generic UC View, we use Com(A) here. For an agent A, we identify the domain of event S (a subset of ACT); we further assume the set *S* is divided into disjoint sets S_{prov} (inputs, events on provided interfaces) and S_{req} (outputs, events on required interfaces).

The notation uses high-level abstractions representing specific sequences of atomic actions to provide much higher "expressive power" with respect to readability of protocols. In particular, procedure calls (inherently non-atomic) are modeled with a pair of atomic events; a call of an operation *op* is captured with atomic events representing *request* (*op*↑) and *response* (*op*↓). Thus, sequence *?op*↑ **;** *!op*↓(**;** is the operator for sequencing) models receiving call of the operation *op* as absorbing a request event for *op* and emitting a response event. Conveniently, the shortcuts *?op*, *!op*, and *?op{Prot}* can be used for sequences *?op*↑ **;** *!op*↓, *!op*↑ **;** *?op*↓ and *?op*↑ **;** *Prot* **;** *!op*↓ respective. This way, it can be expressed that a sequence of events occurs during a procedure call, while the higher level abstraction of the procedure call is still preserved. This significantly improves the readability of the notation. The events *a*↑ and *a*↓ are atomic, while the shortcut abstractions are nonatomic, embracing operation call-like pairs of events.

A behavior protocol $Prot^A$, syntactically generates a set of traces over ACT* (denoted $Com(Prot^A)$, conveniently a regular language). Employing a regular expression-like notation, behavior is described using event tokens of events from ACT and the following operators (given in priority order): ***** (repetition), **;** (sequencing), | (parallelism, based on arbitrary interleaving of the atomic events captured in a pair of traces generated by A and B), || (parallel-or, shortcut for A + B + A|B) and finally **+** (alternative),. Further, the composed

operators (taking a set $X \subseteq ACT$ as their third operand) are *composition* ($\sqcap_X$), *adjustment* ($|_X|$) and *consent* ($\triangledown_X$). To give an example, a simple behavior protocol $Prot^A$ = *?init* **;** *!query****;** *?done* specifies that agent *A* first absorbs the event *init*, then emits an arbitrary number of *query* events and eventually absorbs a *done* event (figure 4.2 on page 26 demonstrates a more elaborate example).

Composition $A \sqcap_X B$ yields the behavior resulting when agents described by protocols A and B are composed together; X is the set of event tokens for events transmitted between these agents. A pair of traces from Com(A) and Com(B) is arbitrarily interleaved, except for occurrences of *?x !x* (or *!x ?x*) in the resulting trace ($x \in X$), which are replaced by τx (an internal action). All occurrences of events from X have to be processed this way; pairs of traces where the events from X do not match (via ! / ? correspondence) are not included in the resulting behavior.

The adjustment operator ($A |_X| B$) also interleaves traces from Com(A) and Com(B), but exact match (not ! / ? correspondence) of events from X is required and, in a way similar to composition, only pairs of traces that match on events from X are included in the resulting behavior.

The consent operator ($A \triangledown_X B$, introduced in [1, 2]) is similar to the composition operator, but generates *erroneous* traces for situations when interaction of A and B results into an error. The types of errors considered are *BadActivity* (A emits *a* but B is not ready to absorb *a*), *NoActivity* (similar to a deadlock situation) and *Divergence* (interaction of A and B never stops). The consent operator implicitly provides a relation for checking the composition of A and B, by considering the composition to be correct if $A \triangledown_X B$ contains no erroneous traces. It is important to note that even with the additional operators, the language generated by a protocol remains regular. For a discussion of related issues and a formal definition, please refer to [70, 1].

Behavior compliance is defined on behavior of a protocol (the language generated by the protocol); the definition is also extended to the protocols. We say that Com(A) is compliant with $Com(Prot^A)$ on set $S \subseteq ACT$ if Com(A) can respond to any sequence of inputs dictated by $Com(Prot^A)$ and for such inputs, creates only outputs anticipated by $Com(Prot^A)$. A formal definition is provided via the adjustment operator:

Com(A) *compliant with* $Com(Prot^A)$ iff:

(i) $Com(Prot^A)/S_{prov} \subseteq Com(A)/S_{prov}$

and

(ii) $Com(Prot^A)/S_{prov} |_{Sprov}| Com(A)/S \subseteq Com(Prot^A)/S$.

Condition (i) requires that the behavior Com(A) restricted to input events contains all input traces dictated by the protocol, i.e., A can accept any sequence of inputs dictated by $Com(Prot^A)$.

In condition (ii), by adjusting Com(A)/S with $Com(Prot^A)/S_{prov}$ (the dictated inputs) over S_{prov}, only traces from Com(A)/S with inputs dictated by $Com(Prot^A)$ are considered; these traces must be contained in $Com(Prot^A)/S$. For reference, the original definition of behavior compliance, with detailed explanation and discussion, is available in [70].

4.2. Pro-cases: An instance of Generic UC View

with minor modifications.

In [70] (and as described in the previous section), behavior of an entity (called agent in the abstract model of [70]) is modeled as a set of traces (finite sequences of atomic events), capturing the communication on the entity's connections (both internal and external). A regular expression-like notation is used to approximate the

actual behavior of entities by regular languages; these concepts are applied to the SOFA hierarchical component model [66, 70]. As the behavior is formally defined and there are powerful operations upon the protocols and behavior (languages), decidable relations and operations exist ($\sqcap_X$, compliant with, consent [1]) which allow to decide on compatibility of two components (their specifications). We show, that (and how) the behavior protocol concept fits into the generic UC view (a key idea here is that a "use case" corresponds to a "protocol").

Basic concepts. An entity exchanges atomic events with other entities on its external connections. In case of an entity S composed of entities A_1, A_2, the connections among A_1, A_2 are internal connections of S, and the events on these internal connections are internal events of S. Other external connections of A_1 and A_2 are the external connections of S. Finally, S is the scope of both A_1 and A_2.

Scenarios. A scenario (called a *trace* in behavior protocols) is a finite sequence of atomic events. The events are denoted by event tokens from a domain ACT in [70]. For our purpose, we assume ACT* corresponds to the Scenarios domain. The subscenario relation is thus defined via a subsequence relation (employing correspondence of !/? to internal events τ) and is therefore decidable. An event is modeled as an event token *a* either emitted (*!a),* absorbed (*?a)* or internally processed (τa). Com(A) $\subseteq$ ACT* denotes the behavior of an entity A (a language upon ACT).

The following example suggests (in a simplified form), how the scenario generated by a use case could be mapped to a trace of a behavior protocol.

<?sic.submitItem, τValidateItem, ?sic.submitPrice, τValidateSeller,
τVerifySellerHistory, !tradecom.validate, τListOffer, !sellernotify.putAuthNr>

The trace corresponds to the main success scenario specification of the use case shown in Fig. 1.1. The actions performed internally by SuD are represented as internal actions (τ), the actions performed by an actor toward SuD are captured as absorbed (by SuD, *?* used), actions performed by SuD toward an actor are captured as emitted (*!*). The event token domain ACT contains names composed of a connection name (e.g., *sic*) and event name (*submitItem*).

Use cases and relations. A *Pro-case* with an entity A as SuD (notation PE^A) is a behavior protocol $Prot^A$ approximating the behavior of A by bounding the behavior of A (see below).

> The original text of [?] (gettingWP) included a brief description of behavior protocols here. We have instead provided a more detailed description in Sect. 4.1; to avoid redundancy, we have omitted the original brief description.

As an example, we show a behavior protocol fragment corresponding to actions 1 and 2 of the sample use case shown in Fig. 1.1 (the extensions and variations pertaining to these lines are considered). The events corresponding to the main success scenario specification are printed in **bold**.

?sic.submitItem { **τValidateItem** ; (**Null** + τPriceAssessmentAvailable ; !sellernotify.putPriceAssessment + τInvalidItem)
}

Here, the *?sic.submitItem{}* shortcut is used to reflect that the action step 2 is performed within (as a part of) the action step 1. The extensions and variations attached to the action step 2 are captured as alternatives (+). In the main success

scenario specification, no additional processing is done after performing action 2 (Null). The condition of the extension (resp. variation) is expressed as an internal action (τ*PriceAssessmentAvailable* and τ*InvalidItem*); this represents the internal choice to be performed by the entity Marketplace Information System.

The language generated by this protocol fragment contains three traces:

1. *<?sic.submitItem↑, τValidateItem, !sic.submitItem↓ >*
2. *<?sic.submitItem↑, τValidateItem, τPriceAssessmentAvailable, !sellernotify.putPriceAssessment↑, ?sellernotify.putPriceAssessment↓, !sic.submitItem↓ >*
3. *<?sic.submitItem↑, τValidateItem, τInvalidItem, !sic.submitItem↓ >*

For a protocol $Prot^A$, $Com(Prot^A)$ denotes the set of traces generated by $Prot^A$. As $Com(Prot^A)$ is a regular language, but Com(A) is not in general, $Prot^A$ only approximates behavior of A. In [70], the approximation is based on the *bounded behavior* relation, A is bounded by $Prot^A$, if Com(A) is compliant with $Com(Prot^A)$. As defined above, Com(A) is compliant with $Com(Prot^A)$ on a set $S \subseteq ACT$ (S being divided into inputs S_{prov} and outputs S_{req}) if Com(A) can respond to any sequence of inputs dictated by $Com(Prot^A)$ and for such inputs, it creates only outputs anticipated by $Com(Prot^A)$. Formally, via the adjustment operator:

(i) $Com(Prot^A)/S_{prov} \subseteq Com(A)/S_{prov}$

and

(ii) $Com(Prot^A)/S_{prov} |_{Sprov}| Com(A)/S \subseteq Com(Prot^A)/S$.

The includes relation is defined as an inclusion of behavior protocol specifications (similar to macro substitution, naturally acyclic); the subscenario preservation property is implied from the definition of Pro-cases.

Whole picture behavior. Typically, only a single Pro-case is used (as the representative of UM^A); such a Pro-case is called the *frame Pro-case* (inspired by *frame protocol* in SOFA [70, 66]). The basic and parallel operators used in the behavior protocols notation can be advantageously employed as the operations for use case expressions (the UEop set); thus, assembling the behavior via use case expressions is natural here.

Addressing consistency issues. Even though the variety of the behavior protocol operators provides strong expressive power (strong enough to describe concurrency, procedure calls, etc.), the behavior (language) generated is a regular language. This significant advantage allows for comparing behavior described by behavior protocols, as, e.g., inclusion of regular languages is decidable. The composition operation $\sqcap_X$ conforms to the generic view of composition (from Generic UC View). The relations compliant with and consent are defined and are decidable, thus the consistency issues (a), (b) and (c) are addressed here.

4.3. Obtaining Pro-cases from Textual Use Cases

In this section, we show a way to transform a textual use case model UM^A into a Pro-case model PM^A (to emphasize the difference between the Generic UC view and its instance Pro-cases, we write PC^A, PM^A, PE^A, and P^A instead of UC^A, UM^A, UE^A, and U^A).

We demonstrate an analogy between a behavior protocol and a use case; based on the similarities identified, we propose how a textual use case can be transformed into a Pro-case.

In this section, we only suggest the guidelines for manual transformation; in the subsequent Chapter 5, we show how this transformation can be achieved in an automated way.

We illustrate the proposed transformation on the Marketplace textual use case model. Based on its scope diagram (Fig. 1.3), we identified the corresponding future software components and their interfaces (Fig. 4.1).

The basic idea is that the textual use case models of the entity Marketplace Information System (*M*) and of the nested entities Clerk (*CL*), Computer System (*CS*) and Supervisor (*SU*) will be transformed into Pro-case models PM^{M}, PM^{CL}, PM^{CS} and PM^{SU} where, advantageously, the assembled behavior is expressed via the operations from PEop as a single protocol – a representative Pro-case. Using the composition operation ($\sqcap_{\chi}$), the behavior of the nested entities (*CL*, *CS,* and *SU*) will be composed together and verified for consistency with the assembled behavior of the enclosing entity (*M*). For this purpose, a protocol verifier tool will be employed to evaluate the compliant with relation.

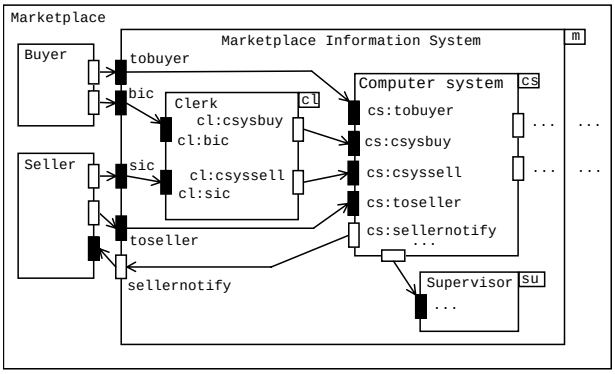


Figure 4.1: Marketplace component diagram

The stages of the whole transformation are:

(i) **Intermediate form**. Based on the textual description of each action (in an action step), we select an event token (roughly: future method) to represent the corresponding action; an event token can contain composed names to reflect the connection names among entities and the action they represent. For example *?sic.submitPrice* is an event token meaning accepting event *submitPrice* on the connection *sic*.

Assuming the textual descriptions use a restricted (simple) form of English (e.g., the "Subject-Verb-Direct object(s)-Prepositions-Indirect object(s)" form (SVDPI) [26]), an action description has typically the form of a request issued by one entity and addressed to another one. We suggest using verb-noun phrases for the action symbols; the connection and the information whether the event is emitted or absorbed is typically obvious from the action step. For the use case "Seller submits an offer" (Fig. 1.1), we get the following intermediate form.

Main success scenario specification:

1. ?sic.submitItem
2. τValidateItem
3. ?sic.submitPrice
4. τValidateSeller
5. τVerifySellerHistory
6. !tradecom.validate
7. τListOffer

8. !sellernotify.putAuthNr

Extensions:

2a τInvalidItem

2a1 Null //Abort

5a τVerifyFailed

5a1 Null //Abort

6a τTradeComValidateFailed

6a1 Null //Abort

Sub-variations:

2b τPriceAssessmentAvailable

2b1 !sellernotify.putPriceAssessment

(ii) **Transforming the intermediate form into a Pro-case**. Applying the sequencing operator ***;*** to action steps is a natural way to transform the main success scenario of an intermediate form into a protocol; however, it is important to identify whether an actions actually spans the execution of several consecutive action step(s) – in such a case, the *a{Prot}* shortcut is to be applied.

Extensions and variations are transformed in a similar way and inserted as alternatives (using +) at their respective positions; their condition is represented as an internal event (τ). Applying these guidelines to our example yields the following Pro-case shown in Fig. 4.2 (walk-through of the main success scenario shown in **bold**):

?sic.submitItem { τValidateItem ; (NULL + τPriceAssessmentAvailable
; !sellernotify.putPriceAssessment + τInvalidItem) **}** ;
(**?sic.submitPrice { τ**ValidateSeller ; τ**VerifySellerHistory** ; (
!tradecom.validate ; (τ**ListOffer** ; **!sellernotify.putAuthNr** +
τTradeComValidateFailed) + τVerifyFailed)
} + τInvalidItem
)

Figure 4.2: Pro-case obtained from the textual use case "Seller submits an offer" shown in Fig. 1.1

(iii) **Assembling behavior**. Based on the set P^A, i.e., several protocols similar to the one above, the Pro-case model PM^A is constructed, by forming use case expressions via operations from PEop.

As an example, consider again UM^M. In addition to what we have presented so far, the whole U^M is to be considered (available in [68]); at this point, listing the names and numbers of the use cases in U^M will do: *#1 Seller submits an offer*, *#2 Buyer searches for an offer*, *#3 Buyer buys a selected item*, *#4 Seller cancels an offer*, *#5 Seller checks on the status of an offer*, *#6 Seller updates an offer*, *#7 Buyer makes a purchase*. In [68], it is easy to see that use case #7 includes use cases #2 and #3. Resulting from that, UM^M contains all these use cases except for #2 and #3, which generate only subscenarios of #7.

At this stage, we assume a PC^M_k has been constructed in the set P^M for each use case #*k* in U^M (we will refer to a Pro-case by the number of the original use case as subscript). In our example, the use cases #1, #4, #5, #6 describe communication of the entity *M* with the same actor, therefore we do not assume any parallelism here and use the + (alternative) operator to assemble these use cases into a sub-

expression (PE^M_1); this actually corresponds to joining their behavior (languages) with ∪ (union). As use case #7 has a different primary actor (the one initiating the generated scenarios), we regard this as source of possible parallelism and we assemble this use case with PE^M_1 via the || (parallel-or) operator (which permits alternative execution, besides interleaved traces). To permit arbitrary iteration, we suffix each of the operands of || with *.

Thus, we represent the behavior of *M* with a single use case expression

$$PE^M_R = (PC^M_1 + PC^M_4 + PC^M_5 + PC^M_6)^* \,||\, (PC^M_7)^*$$

and then construct $PM^M = \{ PE^M_R \}$; obviously, PE^M_R is the frame Pro-case of *M*.

In the appendices of [68], we have provided the whole use case model; for each use case, we have also performed the conversion to a Pro-case. By substituting the individual Pro-cases into the use case expression PE^M_R, we acquire the (expanded) frame Pro-case, which has been included in the appendices of [69] and [67] and is also available in Appendix A of this thesis. Besides [68], the text of the original use cases is also included, in only a slightly modified form, in Appendix B of this thesis (the modifications done are mostly only typo corrections and minor adjustments of style to make the use cases suitable for use with the transformation described in Chapter 5).

(iv) **Compliance test.** Having transformed the textual use case models into Pro-case models which have a representative, we apply the composition operator $⊓_X$ to acquire the composed behavior of the internals of *M*. With the protocol verifier tool [46, 77], we check that the composed behavior of entities *CL*, *CS* and *SU* is compliant with $Com(PM^M)$. This means we check $PE^{CL}_R \,⊓_{X1}\, PE^{CS}_R \,⊓_{X2}\, PE^{SU}_R$ compliant with PE^M_R; here X_1 resp. X_2 is the set of event tokens (roughly action names) on the connections between *CL* and *CS*, resp. *CS* and *SU*.

4.4. Pro-cases – Summary

The proposed behavior specification mechanism Pro-cases interprets all the GUCV abstractions, providing a proof of the concept for the Generic UC View. Namely, it explicitly defines the set *Scenarios*, including its internal structure. As the set *Scenarios* is formed by traces, the subscenarios relation is also defined and decidable. Further, operations for behavior assembly are defined upon traces, and there are "native" interpretations of these operations in the Pro-case notation, which permit to construct a representative use case of a use case model, reflecting the whole picture behavior of the model. The relations defined for Pro-cases address all the consistency issues articulated in Sect. 2.3. In addition, a way to transform a textual use case model into a Pro-case has been proposed. Thus, Pro-cases address the goal (iv) of this thesis.

As Pro-cases are based on in principal regular languages, all the relations and operations introduced in Generic UC View are decidable. In summary, the benefits of creating a Pro-case model are:

(I) With use case expressions, behavior can be assembled to form a single Pro-case as the "whole picture" of an entity's behavior, in a readable and comprehendible notation;

(II) Parallelism can be captured, creating a more precise specification;

(III) Using the decidable compliant with relation, the consistency issues (articulated in Sect. 2.3) (a), (b), (c) can be addressed;

(IV) Pro-cases can be helpful in an early stage of design – showing the interactions of an entity, they can be useful when assigning responsibilities to classes, identifying operations, behavior (structure) of methods; thus, Pro-cases can serve as a powerful replacement of system sequence diagrams [40].

Chapter 5

Converting Textual Use Cases into Pro-cases

natlang
(work in progress)

In the previous chapter, we have outlined the transformation of textual use cases into Pro-cases, considering the conversion only as a manual process. In this chapter, we analyze how existing linguistic tools might be employed to provide a level of automation to this process.

Traditionally, natural language is used for writing use cases. While this makes use cases easily readable to users, it neither permits reasoning on requirement specifications (written as use cases), nor employing the use cases in deriving the initial design of the interfaces of the future system in an automated way. While employing linguistic tools to analyze use cases has already been considered, such attempts usually aim to utilize all the information possibly contained in a use case specification, thus facing the complexity of natural language. Yet, in a use case, the sentence describing a use case step adheres to a simple prescribed structure, and describes an action, which is either a communication action (among entities involved in the use case), or an internal action.

In this chapter, we describe how the principal attributes of the action described by a use case step can be acquired from the parse tree of the sentence specifying the step; we employ readily available linguistic tools for obtaining the parse tree. Having identified the communication actions permits us to construct an (estimate of) behavior specification of the entity modeled by a use case model (in the form of a Behavior Protocol [70]), as well as collect the list of operations accepted and requested by the entity; this may aid with defining the entity's interfaces in an initial design.

5.1. Introduction

5.1.1. Textual Use Cases

Traditionally, natural language is used for describing the actions taken in a step of a use case. Here, the main motivation for using natural language is to make use cases readable and comprehendible to a wide audience, including users of the future system. Figure 5.1 shows a textual use case, following the template proposed in [13]. The use case "Seller submits an offer" has been already shown earlier in Fig. 1.1; however, the phrasing of the use case steps had to by slightly changed to comply with the use case writing guidelines (below); thus there are minor differences between this use case and the original version shown in Fig. 1.1.

> Please note that although we have already described the structure of textual use cases in Sect. 1.3, for reader's sake we also include the description here, where it appeared in the original version of this text in [52] (natlang). The following paragraph has already appeared in the introduction in Sect. 1.3.

The most typical scenario of a use case is specified in the main success scenario specification (top-level block), where each step describes an action that contributes to achieving the goal of the use case; steps in a block are labeled with a step number. Alternative successful flows are specified in the sub-variations section; exceptional situations (error handling) are specified as extensions. Both variations and extensions are attached to a step (by its label), start with a condition (guard), followed by a sequence of steps forming a nested block; here, step labels are prefixed by the extension/variation label. A nested block typically ends with a special action – either moving the flow to a step within the enclosing block, or causing the enclosing block to end. (In case no explicit special action is used, the flow implicitly resumes with the next step of the enclosing block). With special actions, extensions can be used to model repetition or skip a part of the enclosing block. Figure 5.1 demonstrates most of the features described here.

Besides laying out the structure of a use case, recommendations for writing use cases [13, 39] also provide guidelines imposing a simple and uniform structure for the sentences specifying the steps of the use case. Thus, although natural language (NLP) processing is a complex and difficult task in general, employing NLP tools to extract information from textual use cases may yield surprising results.

Use Case: #1 Seller submits an offer
Scope: Marketplace
SuD: Marketplace Information System
Level: Primary Task
Primary Actor: Seller
Supporting Actor: Trade Commission
Main success scenario specification:
1. Seller submits item description.
2. System validates the description.
3. Seller adjusts/enters price and enters contact and billing information.
4. System validates the seller's contact information.
5. System verifies the seller's history
6. System validates the whole offer with the Trade Commission
7. System lists the offer in published offers.
8. System responds with a uniquely identified authorization number.

Extensions:
2a Item not valid
  2a1 Use case aborts
5a Seller's history inappropriate
  5a1 Use case aborts
6a Trade commission rejects the offer
  6a1 Use case aborts

Sub-variations:
2b Price assessment available
  2b1 System provides the seller with a price assessment.

Figure 5.1: Textual Use Case "Seller submits an offer"

5.1.2. Linguistic Tools Available

Readily available linguistic tools permit to acquire a parse tree from a natural language sentence. A parse tree captures the structure of the sentence, according to the grammar of the respective natural language (English in this case). In a *phrase structure* parse tree, the leaves reflect the words of the sentence (preserving the left-to-right order), while intermediary nodes represent *phrases* constituting the structure of the sentence. Figure 5.2 shows a parse tree obtained for the sentence of step 1 of the use case "Seller submits an offer" shown in Fig. 5.1. There, the nouns "item" and "description" (together with ".") constitute a *noun phrase* (denoted NP; in this special case, the phrase is a *basic noun phrase*, denoted NPB), which together with the verb "submits" forms a *verb phrase* (VP). The parse tree also shows the headword of each phrase (e.g., the verb "submits" for the verb phrase). The numbers following the phrase label and the headword are the number of sub-nodes and the index of the sub-node containing the headword.

The "S" phrase-type code denotes a sentence, which is the only single phrase subordinate to the top-level element (denoted "TOP"). Additional phrase-type codes used are "PP" for prepositional phrase and "ADJP" for adjective phrase; an exhaustive overview is available in [6].

As a prerequisite to parsing, the part-of-speech (POS) of each word has to be determined (the word type and the role it plays in the phrase structure). There is a choice between several sets of POS tags; the linguistic tools we employ utilize a subset of the CLAWS tagset [11]. In Fig. 5.2, the POS-tag of "submits" is VBZ (verb in the "s" form), "item" and "description" are nouns (NN); "Seller" is a proper noun (NNP). A related task is obtaining the *lemma* (base form) of a word, based on its actual word form and the POS tag. E.g., the lemma of "submits/VBZ" is "submit" (shown in Fig. 5.2).

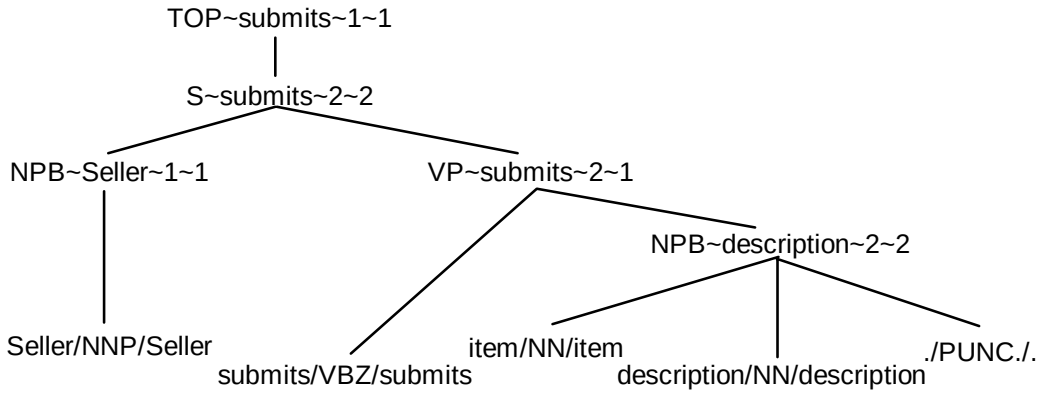


Figure 5.2: Parse tree of the sentence*: "Seller submits item description"*

5.1.3. Goals of this Chapter

In Chapter 4 (and in [69]), we have described transformation of textual use cases into Pro-cases, a notation for use cases based on the formal specification method Behavior Protocols [70]. We proposed guidelines for performing this conversion by hand. In this chapter, we describe how readily available tools for natural language processing can be employed to accomplish this task in an automated way. The key goal is to describe how the simple and uniform structure of sentences used to

describe steps of a use case can be utilized to extract the principal attributes of the action described by the sentence from its parse tree. We achieve this task with only a minimal domain model, consisting only of names of the entities modeled by use cases and optionally of names of conceptual objects.

Subsequently, we explore the options of converting a textual use case into a behavior specification. We demonstrate how a textual use case can be transformed into a behavior protocol, specifying the communication and internal actions of the entity described by a use case model.

This chapter is organized as follows: Section 5.2 describes how to identify the action described by a single step in a use case specification from the parse trees of the sentence describing the step. In Sect. 5.3, we explore employing these results in the conversion of a use case into a behavior specification. We evaluate our approach and discuss open issues in Sect. 5.4. We describe our prototype implementation and practical aspects of our approach in Sect. 5.5; we conclude this chapter in Sect. 5.6. Related work is analyzed in Chapter 8; we outline future work in Sect. 9.3.2 of Chapter 9. The domain model and the version of the use cases actually used in our experiment are included in Appendix B of this thesis.

5.2. Analyzing a Use Case Step

The guidelines for writing use cases prescribe a uniform sentence structure. Readily available linguistic tools allow to obtain a parse tree for such a sentence, showing how the sentence is composed of phrases, types of the phrases, headwords and POS-tags of words (leaves of the tree).

In this section, we show how the type of the action described by a step can be obtained from the parse tree, as well as the principal attributes of the action (i.e., the action type, the communicating actor and the event token). We show how this information can be obtained from the verb, the subject and the direct and indirect object of the sentence (employing the information from the domain model).

We demonstrate our conversion on a use case model describing several entities involved in an electronic marketplace. This use case model has been already used in chapters 1 and 2 (as well as in [69)]; the complete set of use cases is available in [68]. Figure 5.1 shows the first use case of this model (with step 1 slightly rephrased for demonstration purposes). Figure 5.3 shows the scope diagram [13, 69] of the Marketplace system (identical to Fig. 1.3 in Chapter 1, reprinted here for the reader's sake). The domain model we used in our analysis contains entity names: “Seller”, “Buyer”, “Clerk”, “Supervisor”, “Credit Verification Agency”, “Computer System”, “Marketplace Information System”, “Marketplace” and “Trade Commission”; the list of conceptual objects is: “item”, “offer”, “price assessment”, “price” and “payment method”.

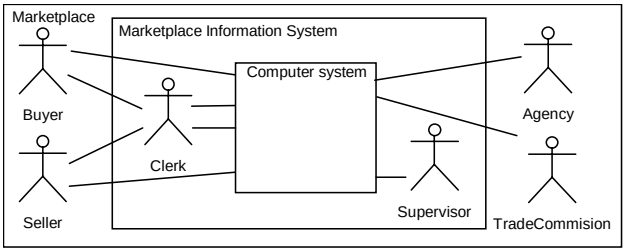


Figure 5.3: Scope diagram of the Marketplace system

5.2.1. Use Case Sentence Structure Premises

Based on the guidelines [13, 26, 39] for writing use cases, we derive the following premises, forming the basis of the conversion described in this paper.

Premise 1: An action described by a step of a textual use case describes either (a) communication between an actor and SuD (a request being sent or information passed), or (b) an internal action performed.

Premise 2: Such action is described by a simple English sentence, adhering to a uniform structure pattern.

Support for these premises can be found in [13, 26], similar recommendations are also found in recent research [21, 25, 24, 38, 44, 65, 73] on natural language use case processing.

In particular, premise 1 is supported by [13], pp. 90-97, where it is required that a step describes *"a simple action in which one actor accomplishes a task or passes information to another actor"*. Further, it is asserted that each step clearly indicates which actor performs the action; the actor has to be the subject of the sentence. In a similar vein, [26] pp. 410 requires that a step describes a *"single atomic task"*.

For premise 2 we find even more evidence. Cockburn [13] specifically requires that *"The sentence structure should be absurdly simple: Subject ... verb ... direct object ... prepositional phrase."*. Graham [26] explicitly defines task action grammar *SVDPI*, imposing sentence structure *"Subject ... verb ... direct object(s) ... preposition ... indirect object(s)"*. Such sentence structure is also prescribed by the controlled language proposed in [73].

5.2.2. Linguistic Tools Used

We employ the well-accepted statistical parser [14, 15] developed by M. Collins at the University of Pennsylvania. While there are several approaches to parsing (rule-based parser, statistical parser), we choose the Collins parser for its high accuracy (over 88% constituent accuracy and over 90% accuracy on dependencies – on generic English text) and for the robustness of its parsing algorithm; also, the Collins parser is using the well-established PennTreebank notation.

We have also considered the statistical parser developed by E. Charniak at the Brown University [10], which also outputs phrase structure trees (but, unfortunately, does not identify phrase heads), and the principle-based parser MINIPAR [45] developed by D. Lin at the University of Alberta (employing a proprietary format).

We use the MXPost tagger [72] for assigning POS-tags (note that with other natural language processing systems, this may be a part of the parser [10]). In addition, we also employ the Morphological tools [56] developed at the University of Sussex to obtain the *lemma* (base form) of the words.

The way these tools were employed in our experiment is described in detail in Sect. 5.5.

5.2.3. Action Type & Communication Information

Step types. From premise 1, we conclude that a step of a use case specifies an operation to be performed, either an *internal action* of an entity, or processing a *request*, either *received* by SuD from an actor, or *sent* by SuD to an actor. Providing support for this conclusion, the well-accepted object oriented methodologies [40]

recommend deriving lists of operations of conceptual objects and system sequence diagrams from (textual) use cases.

Based on premise 2, we assume that the sentence describing a step of a use case adheres to the SVDPI pattern described in Sect. 5.2.1. In such a sentence, subject is the entity performing the action; the verb describes the action. Further, the *direct object* of the sentence describes the data being passed. Moreover, for a request, the *indirect object* of the sentence is the entity receiving the request (if specified). Exceptions to this rule are discussed in Sect. 5.2.5.

The first issue in analyzing the sentence describing a step is to determine the action type. Here, the key lead is the subject of the sentence; in certain cases, the verb is also used.

Subject. Supposing the sentence adheres to the prescribed structure and was successfully parsed by the parser, the main sentence node ("S") of the parse tree contains a *noun-phrase* node and a *verb-phrase* node (in this order). We assume that the first noun-phrase (which may consist of several words) is the subject of the sentence. We match the sequence of nouns ("NN*") in this phrase with the names of entities involved as Actors in the particular use case and with a set of predefined keywords – phrases with special meaning, typically used in textual use cases. The term "System" is frequently used to refer to SuD; in a similar way "User" may be used to refer to the primary actor of the use case (the "Primary Actor" header may designate an entity as the primary actor of the use case). Further, the keywords "Use case" and "Extension" are recognized; these keywords indicate that the step describes a *special action*; such a step will be handled in a special way (discussed in Sect. 5.2.5). A sentence with no subject is also treated as a special action, e.g., "Resume with step 6".

Thus, finding a match for the subject noun-phrase yields the entity active in the step – either the subject is directly the name of an entity, or it is a keyword that in the context of the use case refers to a concrete entity. Note that not having found a match suggests that the sentence was badly written, or, possibly, that the statistical parser failed on the (otherwise correct) sentence; these issues are discussed in Sections 5.2.7 and 5.4.

In case the entity referred to by the subject is an actor involved in the use case, the step describes a request sent by the entity to SuD. Thus, from the point of view of SuD, the step describes receiving a request from the actor.

Example 5.1 In the parse tree in Fig. 5.2, the subject is the name of the entity "Seller" (an actor in this use case); thus the sentence describes a receive action.

Indirect object. In case the subject of the sentence refers to SuD, the step describes either a request issued by SuD toward an actor, or an internal action of SuD. In further analysis, we will look whether the name of an actor is an indirect object of the sentence. For simplicity, we consider as indirect object a noun-phrase subordinate to the main verb-phrase that matches the name of an actor (or the "User" keyword). We consider certain special cases and exceptions, e.g., the actor name must not be in the possessive case, i.e., directly followed by the POS element (marked by an apostrophe). In case the indirect object of the sentence is an actor, the step describes a request sent by SuD to this actor; otherwise, the step describes an internal action.

Example 5.2 In Fig. 5.4, the subject refers to SuD (keyword "System") and the indirect object refers to an entity ("Supervisor"); thus, the sentence describes a

receive action. The step 2 of the use case in Fig. 5.1 (“System validates the description.”) describes an internal action (parse tree shown in Fig. 5.5).

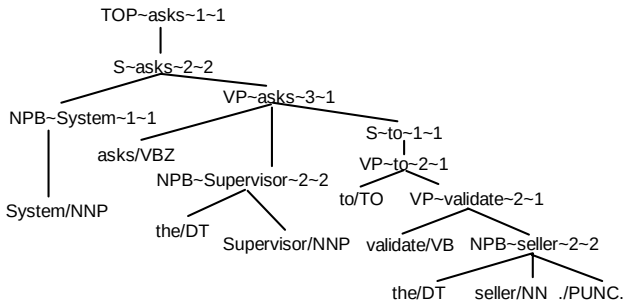


Figure 5.4: Parse tree of *“System asks the Supervisor to validate the seller.”* (step 5 of use case CS1).

When matching the subject and indirect object noun-phrases with entity names, we consider only SuD and the actors declared in the use case header. We choose this approach for simplicity and manageability; alternatively, corrections to use case headers might be proposed when a noun-phrase matches the name of an entity not declared in the use case header.

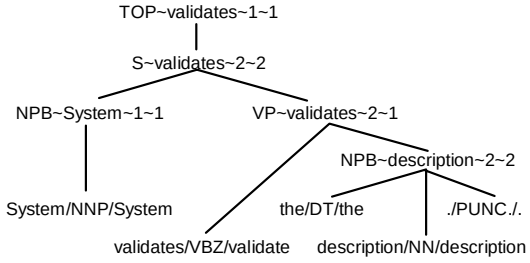


Figure 5.5: Parse tree of the sentence *“System validates the description”*

5.2.4. Estimating Method Call

After determining the type of the action, direction of communication and the actor involved, the next goal is to construct an event token to represent the request in a machine processable behavior specification. Adhering to the widely accepted recommendations and practice to construct method names from verb and object of the sentence (as in system sequence diagrams in [40]), we construct the token from the *principal verb* and the *representative object* description.

Verb. The principal verb is the headword of the topmost verb phrase, unless it is a *padding verb*. In this case, we skip the padding verb and use the next verb in the phrase (possibly nested in a subordinate verb-phrase). For example, let us consider the following sentence: *“System asks the Supervisor to validate the seller.”*(step 5 of use case CS1 in [68]); the parse tree is shown in Fig. 5.4.

This sentence describes a request sent by SuD (keyword “System”) to the actor Supervisor. Here, the verb “asks” is only a syntactical construct, and the verb “validate” actually describes the task requested. We have assembled a list of patterns identifying padding verbs; the list contains word sequences: “ask”, “be”,

“select to”, “choose to”. Note that we compare the words in a pattern by the base form (lemma); matching POS-tag is also required. Also note that we do not consider multiple padding verbs used sequentially in a sentence, as such constructs are not used in practice – and such a sentence structure would clearly be a violation of the use case writing guidelines. On a technical note, we do not apply the padding-verb rule if there are no subsequent verbs in the sentence.

Example 5.3 In Fig. 5.2, the verb “submits” is the principal verb of the sentence; for further use, we consider the base form (lemma) of the verb, i.e., “submit”. In Fig. 5.4, “asks” is a padding verb and “validate” is the principal verb (already in base form).

Direct object. The motivation for acquiring the representative object of the sentence is to obtain words describing the data passed in the request, to be subsequently used in estimating the event token. We obtain the direct object noun phrase of the sentence, and use the relevant words from this noun-phrase. We consider as the direct object noun phrase the first basic noun phrase subordinate to the principal verb; we avoid phrases already identified as the subject and the indirect object. (Technically, we skip a noun phrase if any of its constituent words has already been used – as the subject, the verb or the indirect object. As a fall-back rule when no direct object is found, we try all noun phrases of the sentence).

In this process, we employ the list of conceptual objects (a part of the domain model). In case the direct object noun phrase contains the name of a conceptual object, all words from the noun phrase that are a part of a matching name of a conceptual object are used as the representative object description; otherwise, we use the sequence of all nouns in the direct object noun phrase.

Example 5.4 In Fig. 5.2 (“Seller submits item description.”), the direct object noun phrase contains nouns “item” and “description”. As the list of conceptual objects (part of the domain model) contains the word “item”, the representative object description selected is “item”.

Event token. After the principal verb and the representative object description is identified, we construct the event token; the event token is created from the principal verb and the representative object name. Technically, we concatenate base forms of these words, employing a naming convention to use the first word (the verb) in lowercase, and capitalize the first letter of each subsequent word.

Example 5.5 For the sentence in Fig. 5.4, the verb is “validate” and the representative object is “Seller”, thus, the resulting token is **`validateSeller`**; in a similar way, for Fig. 5.2, with verb “submit” and representative object noun “item”, we get the event token **`submitItem`**.

5.2.5. Special Actions

While most steps in a use case describe communication or internal actions, certain steps are an exception to this rule. Typically used in extensions of a textual use case, some steps describe “meta-actions” changing the flow of the use case. Commonly used patterns of such sentences are “Use case aborts” or “Resume with step <label>”; such patterns are also recommended by the guidelines for writing use

cases [13]. The key point here is that such sentences have a very simple form and are a nearly-constant syntactic construct. In particular, the subject of such sentences is either a predefined keyword (such as “Use case”), or the sentence has no subject at all; moreover, even the verb of the sentence is only one of a few predefined verbs.

We have identified two kinds of special actions, *terminate* actions and *goto* actions. Both are used only in use case extensions and variations and change the flow of control in the *enclosing block*, the block containing the step to which the extension is attached (the main success scenario specification for first-level extensions). By convention, a special action is used as the last step of a use case extension (and only there). A terminate action causes the enclosing block to terminate, while a goto action transfers control to a particular step, identified by a step label.

Terminate. A terminate action terminates the flow in the enclosing block. When the keyword “Use-case” is used as the subject, the action is a *propagating* terminate action and terminates the whole use case; otherwise (when the keyword “Extension” is used), the action terminates only the enclosing block. This distinction applies only when multiple levels of extensions are used; e.g., a terminate action used as the last step of a second-level extension terminates the first-level extension and flow continues with the next step of the main success scenario specification. A terminate action may be an *aborting* action, capturing a failure. In Fig. 5.1, the sentence “Use case aborts.” used in steps 2a1, 5a1 and 6a1 terminates the flow of the use case.

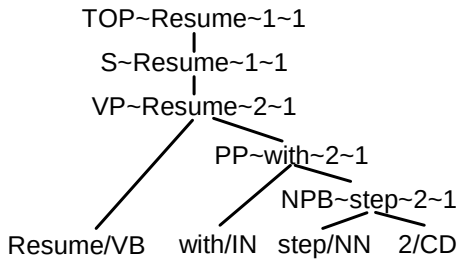


Figure 5.6: Parse tree of the sentence:
“Resume with step 2”

We identify a terminate action when the verb matches on of the predefined patterns: “abort”, “end” and “terminate”. Moreover, the pattern “aborts” also indicates that the terminate action is an aborting action (this may be used in subsequent processing of the extracted information).

Goto. A goto action transfers control to a particular step in the enclosing block. The sentence describing a goto action follows a very simple pattern, solely declaring the transfer of control to a particular step. Such a sentence either has no subject, or the subject is one of the keywords triggering a special action: “Use case” or “Extension”. The verb is one of the following keywords: “continue”, “repeat”, “resume” and “retry” and is followed by a reference to the target step. In this pattern, there is a noun-phrase subordinate to the main verb-phrase (possibly via a prepositional phrase as in Fig. 5.6). The noun-phrase has a “step/NN” and a “CD” (cardinal digit) node; the value of the CD node is the label of the target step of the goto action. Note that semantics of a goto action is not influenced by the subject of the sentence.

As mentioned earlier, special actions are used as the last step of an extension. In case an extension does not end with a special action, we assume an implicit goto action to the step immediately following the step to which the extension is attached; this is common in use case practice (e.g., variation 2b in Fig. 5.1). The use case writing guidelines [13], (p. 108) say that *"Usually, it is obvious [where the story goes]"*. In rare situations, the flow should continue with the step to which the extension is attached, e.g., to retry an action that failed. While it may be obvious to a human reader from the steps of the extension, our approach requires explicitly adding a special action step – which will only improve clarity of the use case.

5.2.6. Conditions of Extension and Variations

The specification of an extension of a use case starts with the extension condition – a textual description of the situation in which the extension applies. We do not aspire to interpret conditions expressed in natural language. While there are clear recommendations for sentence structure of steps of a use case, little can be said about how the extension conditions are specified. In the guidelines [13], the sole recommendation is that *"Condition should describe what the system detects (not what happened)."*; the examples given are "Invalid PIN", "Network is down", "The customer does not respond (time-out)".

Although we do not intend to interpret the conditions, we extract descriptive information from the condition specification to construct a *condition label* that may be used to represent the condition in a behavior specification, and provide a single explanatory identifier for the condition, named in a way a programmer would name, e.g., an attribute representing the condition.

Contrary to our approach to processing use case steps, in this process, we only remove words that do not influence the meaning of the condition specification: prepositions (IN) "of", with", determiners (DT) "any", "the", punctuation (PUNC), the infinitival "to" (TO). However, we preserve words that are essential in meaning of the condition: adverbs (RB) "not", "no", "too" and determiner "no". We also remove padding verbs as described in Sect. 5.2.4. Further, we identify basic noun phrases (NPB) that contain a name of a conceptual object (as in identifying the direct object in Sect. 5.2.4); for such phrases, we use only the words matched, and remove other words of the phrase. Finally, we join all the remaining words into a single label; to preserve the meaning of the condition, we use the actual word forms used instead of the base form (lemma).

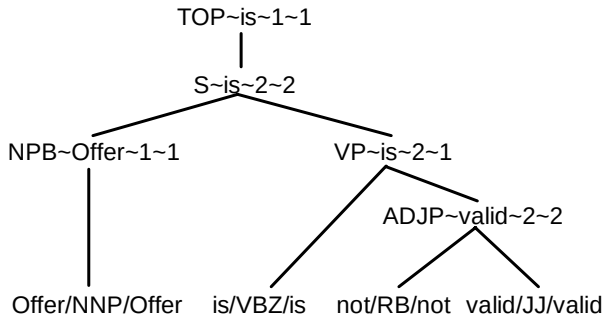


Figure 5.7: Parse tree of condition *"Offer not valid"*

Example 5.6 Let us consider the condition M3-2a “Order is not valid”, with the parse tree shown in Fig. 5.7. The subject noun-phrase matches the conceptual object “Offer”; the verb “is” is a padding verb and is removed, the adverb “not” and adjective “valid” are used, yielding the condition label **`offerNotValid`**.

Note that currently, we aim to translate to behavior specification mechanisms that either do not support conditions, or that model a condition as a black-box internal (condition) event. Thus, we construct the condition label only to play an informative role.

5.2.7. Analyzing Use Case Steps: Summary

In this section, we have demonstrated that by employing the parse tree of the sentence specifying a step in a textual use case specification, it is possible to obtain the type and principal attributes of the action described by the sentence. The action may be an internal action, a communication action (where we also obtain the communicating actor), or a special action. For internal and communication actions, we construct an event token, representing an estimate of a future method name; for a special action, we obtain the type and the target (in case of a goto action). For extension conditions, we construct a condition label as a descriptive representation of the condition. Figure 5.8 shows the automatically obtained event tokens (left), compared to previously manually created tokens (right). The event tokens match on action type and actor identified (except for step 8, relying on context) and provide similar estimates of event tokens.

1 ?SL.submitItem	1. ?sic.submitItem
2 #validateDescription	2. τValidateItem
3 ?SL.adjustPrice	3. ?sic.submitPrice
4 #validateSeller	4. τValidateSeller
5 #verifySeller	5. τVerifySellerHistory
6 !TC.validateOffer	6. !tradecom.validate
7 #listOffer	7. τListOffer
8 #respondAuthorizationNumber	8. !sellernotify.putAuthNr
2a1 %ABORT	**Extensions & sub-variations:**
2b1 !SL.providePriceAssessment	2a1 Null //Abort
5a1 %ABORT	5a1 Null //Abort
6a1 %ABORT	6a1 Null //Abort
	2b1 !sellernotify.putPriceAssessment

Figure 5.8: Action types, attributes and event tokens automatically obtained for the use case “Seller submits an offer” (left) and created by hand in Chapter 4 and in [69] (right)

Discussion on failures. When one of the parts of a sentence is not found (subject, verb, indirect object, direct object), the cause may be either improper structuring of the sentence (too complex, not adhering to the requirement for simple structure), or the statistical parser may have failed at the particular sentence; this is more likely for complex sentences. Adhering to the writing guidelines (which also ask for simple sentences) leads to improved success rate of our tool; this is later discussed in the evaluation in Sect. 5.4.

5.3. Converting Textual Use Cases to Pro-cases

In the previous section, we described obtaining the type and principal attributes of actions specified by use case steps. Here, we describe the second step in converting a use case to a behavior specification. From the wide range of formal techniques (possibly ranging from state machines with transitions labeled by method calls as used in UML [61], to process algebras), we choose Pro-cases [69], based on Behavior Protocols (BP) [70] for the following reasons: (i) same as for use cases, BP are also designed for high readability, (ii) behavior protocols also specify behavior in terms of events sent, received and internally processed (matching our interpretation of use cases) (iii) behavior protocols are designed to specify behavior of software components (matching the use of use cases for nested entities).

Of course, as the event tokens obtained are only an estimate of a future method name, and as we are employing a statistical natural language parser, the resulting behavior specification will be inherently imprecise and only is an estimate of the actual behavior specification, but, regardless that, can be of high value to developers.

5.3.1. Use Cases vs. Pro-cases: Structure

Pro-cases (Behavior protocols) specify behavior in terms of atomic events emitted (!), absorbed (?) and internally processed (τ); the syntax stems from regular expressions. A primary expression specifies a single event; among the operators are `;` for sequencing, + for alternative and * for repetition (and additional operators not used here). Figures 5.12 and 5.13 both show a Pro-case (note that in the automatically constructed Pro-case in Fig. 5.12, "#" is used instead of τ).

Conveniently, the behavior of a Pro-case (a set of finite sequences of events) forms a regular language. This permits reasoning on Pro-case specifications; the compliance relation is decidable and a verifier tool is available [46, 77]. Note that conditions are not supported in Pro-cases, this is actually a trade-off for the regularity of the language. Thus, when converting use cases to Pro-cases, we cannot directly represent conditions of extensions and variations; instead conditions are represented as a *condition event* – a special internal event. The semantics of condition event is, that it may be processed "only when the condition occurs"; in a particular run, once the condition event is processed, it may be processed any number of times. Note that internal events are not considered in the compliance relation (and thus, neither is the condition event).

5.3.2. Creating a Pro-case

We convert the structure of a use case into a Pro-case by first constructing a finite automaton representing the use case, where transitions are labeled with the events as identified in Sect. 5.2; afterwards, we derive a regular expression generating the same language as the automaton.

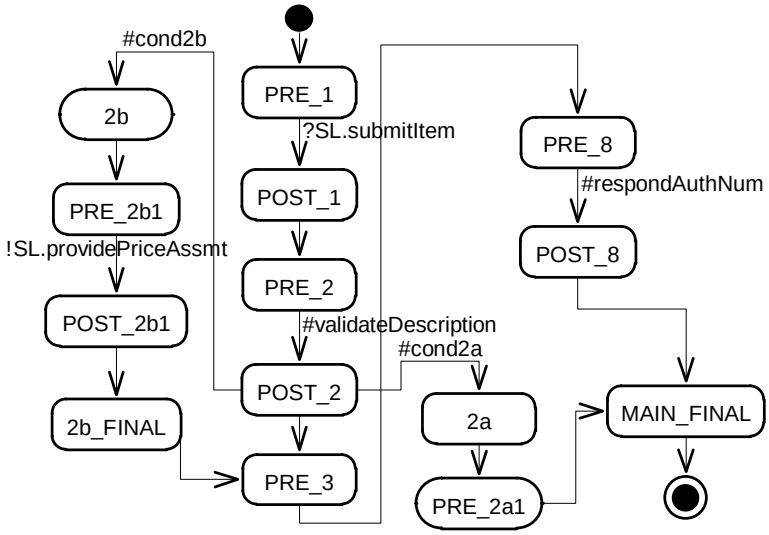


Figure 5.9: Part of the automaton constructed in the conversion of the use case "Seller submits an offer" to a Pro-case

Constructing finite automaton. We start by creating an initial and final state of the automaton, after which we process the main success scenario specification (the top-level block). To correctly capture the flow of a use case (and to avoid introducing new unexpected scenarios), we create a *pre-i* and *post-i* state for each step *i* in a block containing *n* steps; for each block, we also create a *block final* state. Next, we process the top-level block according to the following rules (for simplicity, we also access the initial state as POST-0 and the block final state as PRE-(n+1)):

1. For each step *i* ($1 \leq i \leq n$), we add a *lambda transition* (no event processed) from POST-(i-1) to PRE-i
2. We add a lambda transition from the block final state to the final state.
3. For each step *i* not describing a special action, we add a transition from PRE-*i* to POST-*i* labeled with the event representing step *i*.
4. For each step *i* describing a goto special action, we add a lambda transition from PRE-i to the PRE-state of the target of the step *i.*
5. For each step *i* describing a terminate special action, we add a lambda transition from PRE-*i* to the final state of the enclosing block.

   In case of a propagating terminate action, we process all containing blocks of the enclosing block: starting with the enclosing block and until we reach the top-level block, we add a transition from the currently reached block's final state to its containing block's final state and proceed with the containing block; all these propagation transitions are labeled with the condition event of the innermost extension containing step *i*.
6. For each extension/variation *e* attached to step *i*, we create an *extension initial* state e_{init}, and add a transition from POST-*i* to e_{init} labeled with the condition event of *e*. We consider e_{init} and PRE-(*i*+1) as the initial and final state respective and process the extension block according to the rules 1-6.

We illustrate the construction in Fig. 5.9; the parts shown correspond to a fragment of the main success scenario and the extensions 2a and 2b. The states

“2a” and “2b” are the extension initial states; the *_FINAL states are the final states of the respective blocks.

Deriving a regular expression. We use the generic algorithm for deriving a regular expression (RE) from a generalized nondeterministic finite automaton (GNFA) described, e.g., in [74]. Very briefly, under certain assumptions which hold in our case (no incoming transitions to the initial state and no outgoing transitions from a single outgoing state), the algorithm prescribes that until there is only an initial and a final state, one of the other remaining states (q_r) is removed. In each such step, for every pair of states q_i and q_j (different from q_r), if there is a transition from q_i to q_r and from q_r to q_j, we replace the label of transition from q_i to q_j with the regular expression $e(q_i,q_j) + e(q_i,q_r)\ ;\ e(q_r,q_r)^*\ ;\ e(q_r,q_j)$, where $e(q_k,q_l)$ is the label of the transition from q_k to q_l if there is such, or $\varnothing$ (illegal transition) otherwise. We schematically illustrate the replacement in Fig. 5.10. At the end, the label of the only transition from the initial to the final state is the RE sought.

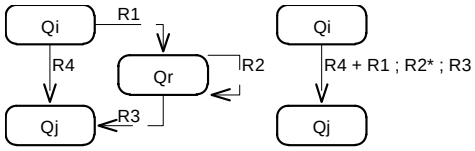


Figure 5.10: Removing a state q_r

Here, the only remaining issue is deciding on the order in which states are removed. The order may significantly influence the form (and length) of the resulting RE (but not the language generated by the RE). With the goal to minimize redundancy in the resulting RE (and thus, enhance readability and manageability), we developed criteria for selecting the state to be removed: for each state, we evaluate redundancy introduced by removing the state; in each step, we remove the state introducing the least redundancy.

```
?SL.submitItem ; #validateDescription ; #cond2a +
?SL.submitItem ; #validateDescription;
(NULL + #cond2b ; !SL.providePriceAssessment ) ;
(  ?SL.adjustPrice ; #validateSeller ; #verifySeller ;
   #cond5a
+  ?SL.adjustPrice ; #validateSeller ; #verifySeller ;
   (  !TC.validateOffer ; #cond6a
   +  !TC.validateOffer ; #listOffer ;
      #respondAuthorizationNumber
   )
)
```

Figure 5.11: Automatically created Pro-case for the use case “Seller submits an offer” (original version as demonstrated in [52] (natlang))

Figure 5.11 shows the Pro-case obtained by employing this algorithm; this has been implemented in the original version of our tool described in [52] (natlang). Enhancing the algorithm has been subject of ongoing research; in Fig. 5.12, we show the Pro-case obtained after implementing additional operations for simplifying the regular expressions. Enhancing the algorithm criteria to efficiently and reliably achieve the least redundancy in the resulting Pro-case remains subject of future research.

```
?SL.submitItem ; #validateDescription ;
(   #cond2a
+   ( NULL + #cond2b ; !SL.providePriceAssessment ) ;
    ?SL.adjustPrice ; #validateSeller ; #verifySeller ;
    (   #cond5a
    +   !TC.validateOffer ;
        (   #cond6a
        +   #listOffer ; #respondAuthorizationNumber
        )
    )
)
```

Figure 5.12: Automatically created Pro-case for the use case "Seller submits an offer" with regular expression simplification employed.

5.4. Evaluation & Open Issues

Evaluation. We have implemented the conversion described here in a prototype tool and evaluated our approach in a case study on a set of use cases published in [68], developed within our previous work [69] (forming the basis of chapters 2, 3 and 4) proposing manual conversion of textual use cases to Pro-cases. With only a minor modification, the use cases were understood by the tool, yielding an estimate of the behavior specification of the subsystems described, as well as the list of events, which may be used in designing the interfaces to the subsystem in the initial design stage.

In Chapter 4 (and in [69]), we have demonstrated the conversion on the use case "Seller submits an offer" (shown also here in Fig. 5.1, with only minor modifications). Figure 5.8 shows the event tokens obtained by our tool (left) and originally created by hand (right) in [69]. Further, in Fig. 5.12, we show the Pro-case obtained with our tool, while Fig. 5.13 shows the manually created Pro-case ([69]). Currently, the prototype implementation does not create the condition labels yet; we use the extension label instead (e.g., "#cond2a"). Also, nested calls are not detected (expressed with curly braces in Fig. 5.13, but not present in Fig. 5.12).

```
?sic.submitItem { τValidateItem ; ( NULL +
   τPriceAssessmentAvailable ;
   !sellernotify.putPriceAssessment + τInvalidItem ) } ;
(  ?sic.submitPrice { τValidateSeller ;
      τVerifySellerHistory ; ( !tradecom.validate ; (
      τListOffer ; !sellernotify.putAuthNr +
      τTradeComValidateFailed ) + τVerifyFailed )
   } + τInvalidItem
)
```

Figure 5.13: Pro-case manually created for the use case "Seller submits an offer"

The modifications of the use cases done were mostly typo corrections. Several sentences in the original use case model did not adhere to the use case writing guidelines (use active voice, write short unambiguous sentences), and had to be corrected in order to produce correct results with the linguistic tools and our prototype tool (simplify sentences, change to active voice).

We intend to perform tests on an industrial use case specification; however, such specifications are usually considered as confidential and are hard to obtain.

Lessons for use case writers. From the case study already done, we have learned that for textual use cases to be machine-processable, the generally accepted use case writing guidelines have to be adhered to. In particular, the sentences have to be simple, describing only communication between SuD and an actor (or an internal action), and use of synonyms has to be avoided. In addition, it is necessary to avoid relying on context (from previous steps), even where the context would be obvious to a human reader.

Thus, use case writers have to learn to write machine-processable textual use cases, but doing so will result into more readable, clear and less ambiguous use cases.

Open issues. While our tool provides satisfactory preliminary results, certain issues remain open. In particular, enhancing the construction of event tokens to yield the same event token for complementary send / receive actions in use case models of communicating entities. Also, at least minimal support for (pre-declared) synonyms would be helpful.

The current implementation does not support including another use case. While this can be done as “macro substitution” (upon the automata), the issue of how to properly handle a failure of the included use case remains open. This triggers another open issue: how to handle failure in Pro-cases; extensions to the underlying behavior protocols specification mechanism are being investigated.

As use cases have a primarily informative goal, the execution semantics of use cases is given only by general consensus. Thus, another open issue is whether multiple extensions attached to the same step may be executed in a sequence; additionally, it is not clear how to identify an extension/variation that is executed *instead* (and not after) the action of the step.

5.5. Implementation: the Procasor Tool

We have implemented the transformation described in this Chapter in the Procasor tool. With readily available linguistic tools, parse trees annotated with lemmas can be obtained for sentences specifying the steps of a use case. Such parse trees are then processed by the Procasor tool, which extracts the principal information of the step actions from the parse trees and converts them into a behavior specification in the form of a behavior protocol. The scheme in Fig. 5.14 shows how the tools are linked together in our prototype implementation. In Sect. 5.5.2, we describe in detail how the linguistic tools are put together to obtain the annotated parse trees; in Sect. 5.5.3, we describe the Procasor tool and the results it provides. Appendices B, C and D contain listings of the input files processed by the linguistic tools, the annotated parse trees and the resulting data obtained with the Procasor tool.

5.5.1. Case Study: the Marketplace Example

We have performed a case study on the set of use cases describing a sample marketplace system, developed during our previous work on Generic UC View. The set of 19 use cases, available in [68], describes behavior of four entities: Marketplace Information System, Computer System, Clerk and Supervisor.

Although the use cases were originally written in a word processing program, for use with the prototype tool, we have converted them into a purely textual file. We separated use case headers from step descriptions; the step descriptions file was processed with the linguistic tools, while the use case headers were only used at the later stage by the Procasor tool. In our prototype implementation, we decided to do this step manually; the time cost was approximately 30 minutes. Assuming certain syntactic rules are adhered to while writing the use cases, this task can be automated in a fully-fledged implementation, either as a macro in the word processing program, or as a script when the use cases are written in a purely textual file.

In the step descriptions file, each line describes a single use case step and has a label identifying the step and the use case it belongs to. The step descriptions file and the headers file used in the example are in Appendix B; only minor modifications (mostly typo corrections) were done to the use cases as originally published in [68].

5.5.2. Employing Linguistic Tools: Detailed Setup

We illustrate the scheme of our prototype implementation and the way it employs the existing linguistic tools in Fig. 5.14. The Collins parser [14, 15] we decided to use requires its input to be carefully prepared; we do this in a number of stages:

(i) We start with initial pre-processing, which includes replacing parentheses "(" and ")" with the symbols "-LRB-" and "-RRB-" respectively, as well as stripping of the step labels and references to included use cases.

(ii) Subsequently, we employ a *tokenizer* to separate standalone grammatical elements (such as punctuation) with white-space; we use the tokenizer developed as a part of the EGYPT project [84] at Johns Hopkins University.

(iii) Next, we use the MXPost tagger [72], developed by Adwait Ratnaparkhi at the University of Pennsylvania, to assign part-of-speech (POS) tags to grammatical elements identified by the tokenizer. The tagger operates with only a limited context and may thus not be correct in ambiguous cases, however, although the POS-tags are used by the parser, the parser may (and does) override the POS-tags assigned by the tagger.

(iv) Finally, we use a simple perl script to add the count of tokens in the sentence to the beginning of each line; this completes the preparation of the parser input.

For the sentence used in Fig. 5.2, the desired input form is:

```
5 Seller NNP submits VBZ item NN description NN . .
```

We process the prepared input file with the Collins parser [14, 15]; for each input line, the parser outputs the most probable parse, accompanied with probability of parses of phrase structures within the parse tree. Currently, we do not utilize the probabilities; from the parser output, we only take the single most probable parse tree produced for each sentence.

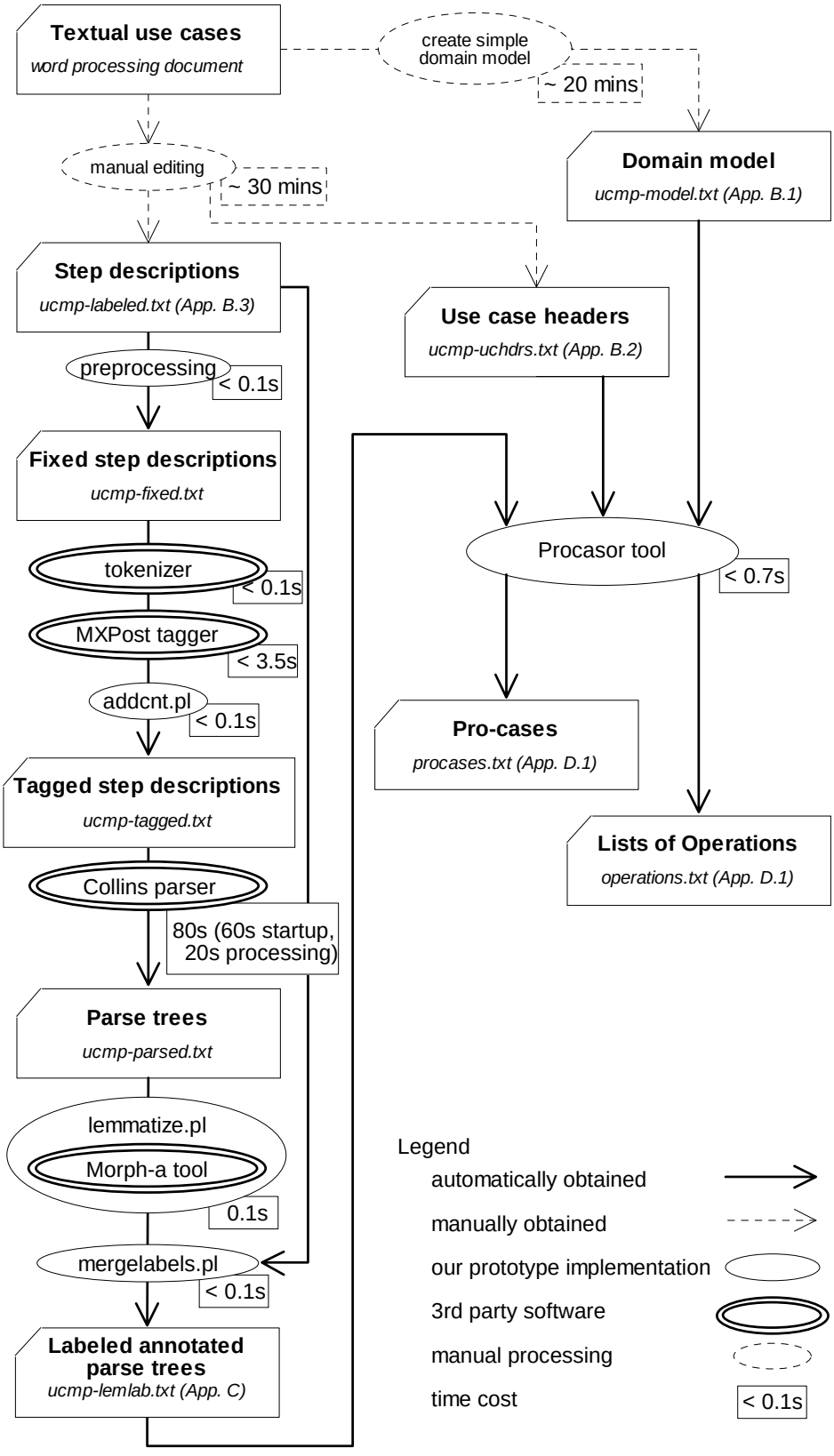


Figure 5.14: Prototype implementation scheme

We subsequently use the Morphological tools suite [56], developed by John Carroll et al. at the University of Sussex, to obtain the lemma for each word in each parse tree. In the parser output, each leaf of a parse tree (a word in the sentence) is represented as a tuple *word/POS-tag*; we add the lemma as a third element, also separated with the slash character. We now re-insert the step labels into this annotated parser output; the form now obtained is used as input for our Procasor tool.

Appendix C lists the data obtained for the Marketplace example. In the `ucmp-lemlab.txt` file listed there, each line represents a single parse tree, starting with the step label (same as in the step descriptions input file ucmp-labeled.txt), followed by the parse tree; parentheses are employed to express the tree structure. Each non-leaf node of the tree starts with a four-tuple (separated with the tilde character "~") consisting of the phrase type code, the head word of the phrase, the number of subordinate nodes and the index of the subordinate node containing the headword. E.g., `VP~submits~2~1` denotes a verb-phrase containing two subordinate nodes; "`submits`" is the headword of the phrase and belongs to the first subordinate node. A leaf node is a three-tuple of the form *word/POS-tag/lemma*.

In the scheme in Fig. 5.14, we show the time cost associated with each step. The values were measured on a Dell Precision WorkStation 530 MT(2x 2.2GHz Intel Xeon processor); they are provided only to illustrate the complexity of the tasks. For most of the tasks besides the parser, the time cost is minimal; even the tagger completes within 4s for the whole use case set. While the time cost for the parser is rather high, most of the time is the initialization of the parser. For the actual parsing, the duration was only 20s for the 178 sentences of the use case model; that averages to less than 0.12s per single sentence. Assuming the startup cost can be eliminated by keeping the parser initialized, the performance results suggest that the transformation can be integrated into a use case editing tool, where the estimated event label can be displayed instantaneously after a use case step is entered; in a similar way, the Pro-case estimate can be updated while the use case is being edited. In Fig. 5.15, we show how the linguistic tools and parts of our prototype implementation might be integrated into an envisioned use case editing tool. We further discuss future work options in Sect. 9.3.2.

As for licenses, the Collins parser and the EGYPT project are distributed under the GNU General Public License (GPL); the MXPost tagger and the Morphological tools suite are distributed under a license that permits personal use for research and education, but requires authors consent with commercial use or further distribution.

5.5.3. The Procasor Tool and Results

For the transformation described in this chapter, the Procasor tool requires a simple domain model, listing the entities involved in the use cases; further, to aid with identifying the representative object of a sentence, the domain model may contain a list of conceptual objects. In our prototype implementation, we created the domain model by hand; as even a simple model suffices, the associated time cost was fairly low. Nonetheless, identifying the entities can be automated by processing the use case headers; identifying the conceptual objects remains a task to be done manually.

Appendix B.1 shows the domain model describing the Marketplace system. For each entity, a line started with the **actor** keyword defines the identifier used for the entity and the entity name (the name may consist of multiple words). In a similar vein, the conceptual objects, marked with the keyword **domainname**, contain an internal identifier and the list of words constituting the name of the conceptual object. Finally, the domain model refers

to files contain the use case headers and the annotated parse trees of sentences of the use cases.

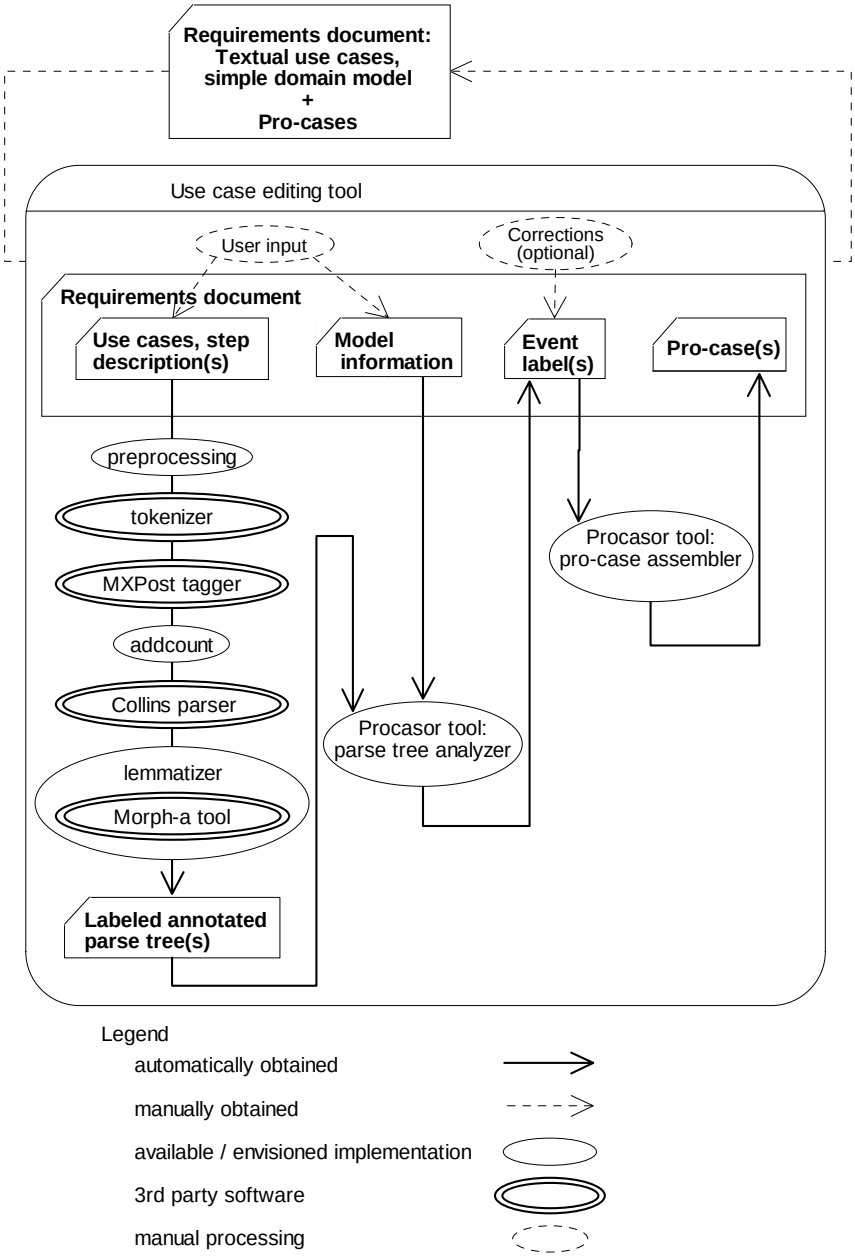


Figure 5.15: Scheme of envisioned integrated use case writing tool

Based on the annotated parse trees (App. C) and the domain model (App. B.1), the Procasor tool for each use case step identifies the action type, the involved actor (if any) and creates an estimate of the event label, as described in Sect. 5.2 earlier in this chapter; the event tokens constructed fore each step are listed in Appendix D.1. After processing all steps of a use case, the tool constructs a finite automaton and converts it into a Pro-case, according to Sect. 5.3. The Pro-cases created by the tool are shown in Appendix D.2; moreover, to illustrate the future potential of employing the approach in a CASE tool, the Procasor tool also outputs the list of events (operation requests) to be received, sent and internal processed by each entity, shown in Appendix D.3. To illustrate how the automatically created Pro-

cases relate to the ones created manually in our previous work proposing Pro-cases [69], we have (manually) created a frame Pro-case, shown in Appendix D.4, by substituting the automatically created Pro-cases into the use case expression already assembled for the Marketplace use case model; the original frame Pro-case constructed from the manually created Pro-cases is shown in Appendix A.

5.5.4. Linguistic tools: Alternatives

In our exploration of linguistic tools available, we have also considered the parser developed by Eugene Charniak at the Brown University [10] and the MINIPAR parser developed by Dekang Lin at the University of Alberta [45]. We eventually selected the Collins parser for its high accuracy and robustness. However, using other parser would simplify the tool setup scheme as, e.g., the Charniak's parser encompasses a tokenizer and tagger and requires only a minimal preparation of the input file (enclosing each sentence into an XML element).

5.6. Conclusion (Chapter Summary)

We have described a way to convert a textual use case specification into a behavior specification by employing linguistic tools and by identifying the type and principal attributes of the action described by a use case step based on the parse tree of the sentence specifying the step. In the subsequent stage, the structure of a use case is converted into a Pro-case (employing the Behavior Protocols notation [70]). The conversion described in this paper is implemented in a prototype tool, providing reasonable accuracy on recognizing the action type and communicating actor; the accuracy of estimating the event token is hard to evaluate. By employing the uniform structure of sentences in a use case specification, it is possible to define a conversion scheme utilizing only the principal information contained in a use case specification and to construct a behavior specification.

Thus, it is possible to write use cases in natural language, readable to a wide audience, and enjoy at least some of the benefits of formal methods at the same time.

Chapter 6

Analyzing Use Cases in UML 2.0

uml20uc
(work in progress)

In Chapter 3, we have analyzed the Unified Modeling Language (UML) in the version 1.4/1.5, currently used by the software engineering industry. While UML 1.5 is still the current official standard, the emerging standard UML 2.0 is reaching completion. UML 2.0 features four behavior specification mechanisms that may be used for use cases: ***Interactions***, ***Activities***, ***State Machines*** and ***Protocol State Machines***. The key goal of this chapter is to evaluate whether and how these behavior specification mechanisms address the issues articulated in Sect. 2.3. To capture the diversity among these specification mechanisms, we extend the *Generic UC View* model introduced in Chapter 2. The main motivation for these extensions is to utilize the features of a behavior specification mechanism where semantics is defined based on traces, while preserving support for other behavior specification mechanisms as well. We address this issue by introducing two specialized extensions of our generic model, *Basic UC View* not concerned with the trace semantics and *Trace-based UC View* particularly tailored for behavior specification mechanisms with trace semantics.

Utilizing the properties of these two models, we refine the assembly operations envisioned in the Generic UC View as the operations (from set UEop) to be used in use case expressions. Further, as the structure and syntax of the behavior specification mechanisms available in UML 2.0 is well defined, we explore the options to interpret the assembly operations via native operations of these behavior specification mechanisms.

In this chapter, we analyze UML 2.0 [61] in the version published in OMG document ptc/03-08-02: Unified Modeling Language: Superstructure, version 2.0, Final Adopted specification. The specification is being scrutinized by the OMG UML 2.0 Superstructure Finalization Task Force (FTF) at this time. As of May 2004, the FTF has released an interim report [63] and also, as a convenience document, an updated version of the UML 2.0 specification in ptc/04-05-02 [62]. Some of the issues identified there coincide with our findings (in particular OMG Issue 6445, see Sect. 6.6.5). We have examined the changes and updates made in [62]; the parts of the specification we focus on did not change significantly and the issues we have identified apply to [62] as well. Nonetheless, for the sake of precision, [61] was the specification used in our analysis.

6.1. UML 2.0 Overview

The new version of the Unified Modeling Language (UML) preserves the language actually used by its users, however, it significantly changes the way the language is defined and the core it is based on. The key motivations for developing UML 2.0 indeed were to re-develop the language on a soundly based core; additional motivation was to align UML with MOF, so that they are based on the same core. For manageability, UML 2.0 splits the language definition into four parts: Infrastructure, Superstructure, OCL and Diagram Interchange.

Infrastructure. The Infrastructure specification provides the kernel of the language; the intention is to share this kernel with the upcoming MOF 2.0 specification. The Infrastructure specification rebuilds UML in a modular way, so that implementations can choose various levels of compliance. Most metaclasses ***Element***, ***Classifier***, ***Class***,...) are declared in multiple packages, in each such package, the metaclass declaration specializes extends the previous declaration with additional features.

Moreover, the Infrastructure specification also manages UML extensibility, either via profiles (as in UML 1.x), or, in a more powerful way, using MOF to subclass the UML metaclasses.

In general, the UML 2.0 Infrastructure re-engineers the UML core, preserving the semantics and usage, but doing so in an internally more consistent way. The improvements also include separation of concerns between semantics and notation.

Superstructure. The Superstructure specification defines the UML 2.0 language based on the Infrastructure. In addition to the concepts already present in UML 1.x, UML 2.0 features new metaclasses ***StructuredClassifier*** and ***EncapsulatedClassifier***. Based on these enhancements, the ***Component*** metaclass is defined, reflecting a possibly hierarchical component featuring provided and required interfaces; this corresponds to the concept of a component considered by the architecture description languages (ADLs) such as Darwin [47], SOFA [66] and Fractal [8].

A ***StructuredClassifier*** supports nesting by decomposing the functionality into several ***parts***, connected by ***Connectors***. Each ***part*** represents only a ***Role*** to be played by a concrete instance; the choice of the implementing classifier may be postponed. Thus, the components internal structure is specified with a grey-box view; the “Role” concept corresponds, e.g., to the component *architecture* definition in the SOFA component model.

Besides the structured classifiers, the Superstructure specification also introduces a new generic behavior model. Here, the class ***Behavior*** represents an executable ***Element*** which can exhibit behavior. Via ***BehavioredClassifier***, the ***Class*** metaclass is extended with the ability to own a Behavior, as well as to specify behavioral features available (such as operations) represented by the ***BehavioralFeature*** metaclass (a specification to be implemented by an instance of a subclass of ***Behavior***).

The generic behavior model only specifies which properties of a model are accessible to a Behavior. The concrete semantics have to be specified by subtypes of Behavior; the subclasses provided are ***Interaction***, ***Activity***, ***StateMachine*** and its specialization ***ProtocolStateMachine***. Compared to UML 1.x, StateMachines have been significantly reworked; now, they support inheritance (of state machines) and nested states. Of the two types of state machines considered, behavioral State Machines are intended for implementation specification and Protocol State Machines for usage specification.

UML 2.0 provides support for specifying software architectures based on (nested) component; at all levels of nesting, the components may be amended with behavior specifications. Further, the specification mechanisms of UML 2.0 are explicitly considered as a possible notation for use cases, which creates a particular interest in evaluating these specification mechanisms in this thesis.

6.2. Goal and Structure of this Chapter

Obviously, in order to utilize all the potential of the use case idea, there is the need to precisely define the semantics of the behavior specification mechanism employed in a use case specification.

The emerging standard UML 2.0 features mechanisms to capture relations among use cases as well as to specify behavior of a use case. Therefore, an interesting problem is to which extent these UML mechanisms support behavior assembly, behavior composition, and consistency reasoning. Addressing this problem is the key goal of this chapter (also, the paper [51]).

The chapter is organized as follows: Section 6.4 provides a extensions to the generic model of use cases to provide a framework suitable for analyzing UML 2.0. Section 6.5 addresses the key goal of the paper by analyzing the UML 2.0 behavior specification mechanisms and identifying how the assembled and composed behavior can be constructed in UML 2.0. Section 6.7 contains evaluation; the chapter is summarized in the concluding Section 6.8. Related work is discussed in Chapter 8 and future work is outlined in Sect. 9.3.

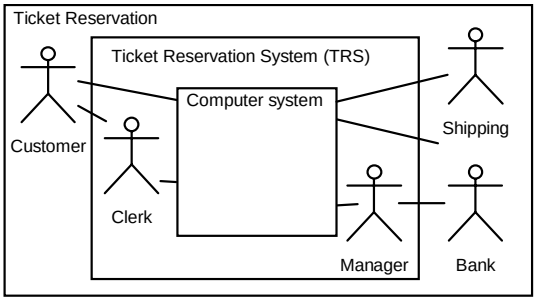


Figure 6.1: Scope diagram of Ticket Reservation

6.3. Behavior Assembly and Composition Illustrated

In this chapter, we illustrate our approach on the example of a Ticket reservation system. Figure 6.1 shows the scope diagram of this system; fig. 6.2 demonstrates a selection of fragments of use case written for the entities Ticket Reservation System (TRS) and Bank.

> Please note that although we have already introduced the concepts of behavior assembly and composition in Sect. 1.4, we also include the discussion here, to preserve the original flow as published in [51] (uml20uc); the text of this section has already used been in a modified form with different example in Sect. 1.4.

For simplicity, we use plain English textual notation, where the scenarios specified by a use case are determined by the *main success scenario specification* showing the "most typical" scenario, and by its *variations* and *extensions*. While the use cases TRS1 and TRS2 are shown only as fragments without variations and extensions, the use case B1 demonstrates how an extension would be specified. In a use case header, *scope* means the parent entity; the other elements have obvious meaning.

Apparently, in addition to Pay for a Ticket and Deliver Ticket, there would be more use cases of TRS – all of them together specifying the behavior of TRS. The scenarios specified by these individual use cases may be required to be performed in a sequence, concurrently, etc.; in general we call the process of "putting use cases together" *behavior assembly* which yields the *assembled behavior* of a SuD. In this view, the *use case model* of TRS is determined by all its use cases and by the way they are combined.

In a similar vein, for composed entities the question is whether it is possible to obtain the behavior of the containing entity by *composition* of the assembled behaviors of the nested entities. The *composed behavior* has to capture the internal communication among the nested entities, as well as those of their activities that are visible outside as the behavior of the containing entity. In the example above, TRS communicates with Bank in such a way that the steps 1 and 2 of B1 correspond to the step 5 of TRS1. Except for this special case (synchronization), in which the complementary communication actions become an internal activity, the Bank and TRS operate in parallel and their activities interleave. The communication of Manager and Bank reaching outside of TRS, gives an example of an externally visible communication.

Use Case: TRS1 (Pay for a Ticket)
Scope: Ticket Reservation
SuD: Ticket Reservation System
Actors: **Customer**, **Bank**
Main success scenario specification:
1. Customer selects to buy the reserved tickets.
2. System validates the ticket reservation.
3. System informs the Customer about the amount to be paid.
4. Customer enters payment information.
5. System validates the payment information with the Bank

...

Use Case: TRS2 (Deliver Ticket)
Scope: Ticket Reservation
SuD: Ticket Reservation System
Actors: **Shipping**, **Customer**
Main success scenario specification:
1. System looks up the Customer's address
2. System validates the address with the Shipping service
3. System send the tickets via the Shipping service
4. System sends the customer a shipment tracking id

...

Use Case: B1 (Validate Payment)
Scope: Ticket Reservation
SuD: Bank
Actors: **Ticket Reservation System**
Main success scenario specification:
1. The bank receives a payment validation request
2. Bank validates the payment and sends response.

Extensions:
2a Validation fails
  2a1 Bank sends notification that validation failed
  2a2 Use case aborts

Figure 6.2: A sample of textual use cases

6.4. Basic and Trace-Based UC View

In Chapter 2 (and [67, 69]), we introduced *Generic UC View*, a generic use case model, mostly inspired by our previous work on behavior protocols designed for behavior specification of nested software components [70, 71]. In this section, we introduce *Basic UC View* (a modification of Generic UC View) and its specialization *Trace-Based UC View*, which allow us to analyze how the behavior specification mechanisms in UML 2.0 address behavior assembly and behavior composition.

> For the sake of readability, besides describing the extensions to the model, we also briefly refresh the basic concepts of the original model used in the model extensions.

6.4.1. Basic Concepts

An entity S composed of sub-entities A_1, ... ,A_n forms the *scope* of each A_i; the topmost scope is called *system*. An entity A_i communicates through *connections* with other (*actors*) A_j of the scope S, and with other external actors located in the parent scope; in this setting, A_i is the SuD. Advantageously, the nesting of entities and their scopes can be expressed as a scope diagram; an entity is represented either by the stick-figure symbol or by a rectangle, a line represents a connection.

Note that a scope diagram captures the relations among entities, not among use cases and entities (as UML use case diagram does). Figure 6.1 shows the scope diagram of a sample *Ticket Reservation* system. The use cases TRS1 and TRS2 in Fig. 6.2 describe the interaction of Ticket Reservation System with its surrounding actors in the scope of Ticket Reservation.

6.4.2. Basic UC View

Scenarios, behavior. Assuming A is an entity in a system Σ, a particular way of activity of A in a run of Σ is captured as a *scenario*. The possible scenarios of A in any run of Σ form the *behavior* of A. In order to accommodate a broad range of systems and behavior specification mechanisms, we denote by $Scenarios_\mu$ the set of scenarios and sub-scenarios reflecting the activities of all possible entities in a considered collection of systems; by the subscript μ we emphasize that the elements of $Scenarios_\mu$ are expressed in a specification mechanism μ. All the scenarios of A expressible in μ form the *μ-behavior* of A denoted $Com_\mu(A)$, $Com_\mu(A) \subseteq Scenarios_\mu$.

Use case. Given an entity A as SuD, we denote by $UC_\mu^{A}{}_i$ the i-th use case specified for A, in which the behavior specification is written in μ. Assuming for simplicity that every use case has exactly one behavior specification associated with it, we unify the use case and the associated behavior specification: $Com_\mu(UC_\mu^{A}{}_i)$ stands for the behavior specified by the use case (i.e. by its associated behavior specification) and $Com_\mu(UC_\mu^{A}{}_i) \subseteq Scenarios_\mu$. All the use cases which can be written in μ for A form the domain U_μ^A, formally $U_\mu^A = \{UC_\mu^{A}{}_i\}$ where i is from a given set of indexes.

Example 6.1 For illustration, consider the textual use cases from Fig. 6.2. By the convention above, we should denote them as $UC_{Text}^{TRS}{}_1$ and $UC_{Text}^{TRS}{}_2$. Obviously, the actual behavior specification (in the specification mechanism "Text") in these use cases is technically the text after "Main success scenario specification:". The set $Scenarios_{Text}$ is defined as the set of all possible sequences of steps of entities shown in Fig. 6.1. Therefore, $Com_{Text}(UC_{Text}^{TRS}{}_1)$ = { < Customer selects to buy ..., System validates .., The Bank ... > }, $Com_{Text}(UC_{Text}^{TRS}{}_2)$ = { < System looks up ..., System validates ..., ... tracking_id ... > }. Note, however, that Fig. 6.2 captures just fragments of the behavior specification so that $Com_{Text}(UC_{Text}^{TRS}{}_1)$ and $Com_{Text}(UC_{Text}^{TRS}{}_2)$ would be much larger in reality (typically due to extensions).

Use case expressions. We intend to define *use case expressions* with the intention to capture assembled behavior – i.e. the behavior specified by several use cases (for a particular A as SuD). Therefore we assume that use case expression are formed by means of *assembling operators* from the set OP = { + ; | * }, denoting alternative, concatenation, parallel composition, and repetition. Our choice of OP is driven by the intention to assemble behavior via a "minimal set" of operations with intuitively clear meaning. For example, if $UC_\mu^{A}{}_1$, $UC_\mu^{A}{}_2$ are use cases, the use case expression $UC_\mu^{A}{}_1 + UC_\mu^{A}{}_2$ means that either A behaves like $UC_\mu^{A}{}_1$ specifies, or A behaves like $UC_\mu^{A}{}_2$ specifies (alternative). In a similar way, $UC_\mu^{A}{}_1 ; UC_\mu^{A}{}_2$ means that A chooses one of the "runs" specified by $UC_\mu^{A}{}_1$ and then it behaves according to $UC_\mu^{A}{}_2$ (sequencing). $UC_\mu^{A}{}_1*$ stands for repetition, i.e., A behaves like specified by $UC_\mu^{A}{}_1$ several times, while $UC_\mu^{A}{}_1 \mid UC_\mu^{A}{}_2$ expresses parallel execution of the behaviors specified by $UC_\mu^{A}{}_1$, $UC_\mu^{A}{}_2$. All use cases in a use case expression have

to employ the same behavior specification mechanism μ; again, the symbol denoting this specification mechanism is used as a subscript, e.g., $e_\mu = UC_\mu{}^A{}_1 + UC_\mu{}^A{}_2$.

Above, we defined the meaning of the operators intuitively. To give an operator op ∈ OP a precise meaning in a specification mechanism μ, we have to associate it with an operation θop: $Scenarios_\mu \times Scenarios_\mu \rightarrow Scenarios_\mu$. This way the behavior described by an expression of the form $Uc_\mu{}^A{}_i$ op $UC_\mu{}^A{}_j$ is $Com_\mu(UC_\mu{}^A{}_i)$ θop $Com_\mu(UC_\mu{}^A{}_j)$. In a similar way, we inductively define the behavior of a general expression of the form $e_{\mu,1}$ op $e_{\mu,2}$ as $Com_\mu(e_{\mu,1}$ op $e_{\mu,2}) = Com_\mu(e_{\mu,1})$ θop $Com_\mu(e_{\mu,2})$, provided $Com_\mu(e_{\mu,1})$ and $Com_\mu(e_{\mu,2})$ were already defined. For illustration, imagine textual use case $UC_{Text}{}^{TRS}{}_1$ and $UC_{Text}{}^{TRS}{}_2$ and the operation ∪ as the operation associated with +. This way $Com_{Text}(UC_{Text}{}^{TRS}{}_1 + UC_{Text}{}^{TRS}{}_2) = Com_{Text}(UC_{Text}{}^{TRS}{}_1) \cup Com_{Text}(UC_{Text}{}^{TRS}{}_2)$ = {<Customer selects to buy ..., System validates .., The Bank ...>, <System looks up..., System validates ..., ... tracking_id ...>}.

Operators interpreted in μ. In general, for a particular μ, it can be impossible to associate an operation from $Scenarios_\mu \times Scenarios_\mu \rightarrow Scenarios_\mu$ with each of the operators in OP. Therefore, we introduce $OP_\mu \subseteq OP$ containing all those operators associated with such an operation.

Example 6.2 By a detailed analysis of the textual use case specification mechanism it becomes clear that it is hard to find a sound semantics for parallel composition of textual use cases (unless some limits are imposed on overlapping of the specified actions); on the other hand finding a "reasonable" semantics for the remaining operators is easy via simple text operations, so that OP_{Text} = { + , ; , * }.

Characteristic use case, native operations. Every behavior specification mechanism μ typically defines native operations upon behavior specifications. In Basic UC View, a native operation $f: U_\mu{}^A \times U_\mu{}^A \rightarrow U_\mu{}^A$ combines behavior specifications of use cases into behavior specification of a new use case; Sect. 6.5 provides a number of examples in this respect. To address behavior assembly, we find it very useful to construct a "summary use case" [13] $UC_\mu{}^A{}_i$ from a set of use cases $\{UC_\mu{}^A{}_k\}$ via native operations by using a use case expression e_μ (its syntax tree) as guidelines. Driven by this motivation, we define $UC_\mu{}^A{}_i$ to be a *characteristic use case* of a use case expression e_μ if $Com_\mu(e_\mu) = Com_\mu(UC_\mu{}^A{}_i)$.

Implementing operators from OP_μ via native operations. In order to construct a characteristic use case $UC_\mu{}^A{}_i$ of e_μ, we have to identify which of the operators from OP_μ have an implementation via native operations of μ.

We say that an operation $f: U_\mu{}^A \times U_\mu{}^A \rightarrow U_\mu{}^A$ *implements* op ∈ OP_μ, if for all use cases $UC_\mu{}^A{}_i$, $UC_\mu{}^A{}_j$ it holds that $Com_\mu(UC_\mu{}^A{}_i$ op $UC_\mu{}^A{}_j) = Com_\mu(UC_\mu{}^A{}_i$ f $UC_\mu{}^A{}_j)$. Here, $UC_\mu{}^A{}_i$ op $UC_\mu{}^A{}_j$ is a use case expression, while $UC_\mu{}^A{}_i$ f $UC_\mu{}^A{}_j$ is a new use case.

If all the operators from OP_μ are implemented by native operations in μ, then, for every use case expression e_μ, a behavior specification (and a use case $UC_\mu{}^A{}_k$) can be constructed recursively by following the syntactic structure of e_μ in such a way that $Com_\mu(UC_\mu{}^A{}_k) = Com_\mu(e_\mu)$. Therefore, the following claim holds:

Claim. If all the operations from OP_μ are implemented via the native operations of a use case specification mechanism μ, then for every use case expression e_μ there can be constructed its characteristic use case.

Example 6.3 Let us implement the operation ";" via a native operation of the use case specification mechanism Text. Such a native operation takes two textual use

cases $UC_{Text}{}^{TRS}{}_{i}$, $UC_{Text}{}^{TRS}{}_{j}$ and creates a new behavior specification $UC_{Text}{}^{TRS}{}_{k}$, consisting of the steps of the main success scenario of $UC_{Text}{}^{TRS}{}_{i}$, followed by the steps of the main success scenario of $UC_{Text}{}^{TRS}{}_{j}$ (the steps are renumbered accordingly). The extensions are modified in a similar way.

Assembled behavior. As mentioned in Sect. 1.4 informally, a use case model of an entity A (SuD) is determined by all its use cases and by the way they are combined (specifying the assembled behavior of A). To capture these concepts formally, we assume that a use case model of A is a pair $<UR_{\mu}{}^{A}, e_{\mu}{}^{A}>$ where $UR_{\mu}{}^{A} \subseteq U_{\mu}{}^{A}$ is the set of relevant use cases ("all its use cases") upon which a use case expression $e_{\mu}{}^{A}$ is written. This expression specifies the *assembled behavior* $Com_{\mu}(e_{\mu}{}^{A})$ approximating the behavior $Com_{\mu}(A)$. If there exists a characteristic use case of $e_{\mu}{}^{A}$, we call it *representative use case* of the use case model.

6.4.3. Trace-Based UC View

To capture behavior composition, the content of the domain $Scenarios_{\mu}$ has to be defined explicitly. To do this, we introduce Trace-Based UC View (an extension of Basic UC View) based on traces. We have chosen this approach because traces are well understood and established in the behavior specification community [5]. To our knowledge, there are no formally-defined behavior semantics (suitable for the problem we are addressing) other than traces, a labeled transition system (LTS) [5] and an algebraic specification (which coexist with LTS as an alternative in defining operator semantics in some process algebras). Semantics defined via LTS and algebraic specifications, on which many well-known formal methods are based (Petri Nets, process algebras – CSP, CCS) have, from the point of view of this paper, features equivalent to trace based semantics. The main difference here is that there is the option to define simulation and bisimulation equivalences on an LTS (or via an algebraic specification), while on traces only trace equivalence can be defined (which is weaker than simulation and bisimulation). However, as we use behavior compliance as our equivalence relation, this is not an issue.

In Trace-Based UC View, an activity of A is a finite sequence of atomic events from a finite domain; these events can be represented by event labels from a domain Act_{μ} (also finite) and a scenario is a *trace* (finite sequence of event labels). Then, $Com_{\mu}(A) \subseteq Scenarios_{\mu} = Act_{\mu}{}^{*}$.

Composed behavior. Let us assume the following setting: An entity C is composed of entities A and B, which communicate via a connection (and via other connection with the environment of C). The goal is to obtain the behavior of C composed of $Com_{\mu}(A)$ and $Com_{\mu}(B)$. A and B perform their activities in parallel; explicit synchronization may occur whenever A and B communicate. The event labels of the actions serving for communication with the entities connected to A form the set $Msg_{\mu}(A) \subseteq Act_{\mu}$.

For entities A and B we define the relation $Pair_{\mu}{}^{A,B} \subseteq Msg_{\mu}(A) \times Msg_{\mu}(B)$ capturing the synchronization of events of A and B, e.g., sending and receiving a message. Any pair of events $t^{A} \in Msg_{\mu}(A)$ and $t^{B} \in Msg_{\mu}(B)$ is *synchronized* if $Pair_{\mu}{}^{A,B}(t^{A},t^{B})$ holds and in this case we call t^{A},t^{B} a *synchronizing pair*. For technical reasons, we denote $Sync_{\mu}(A,B)$ the set of all labels representing events of A that are synchronized with some events of B.

Composed behavior of A and B is captured by interleaving of their traces (reflecting parallel execution) and merging the synchronizing pairs of the events

(internal communication) into new tokens. This way, the composed behavior is the set of all traces resulting from interleaving of each $s^A \in Com_\mu(A)$, $s^B \in Com_\mu(B)$ in the following way:

(1) Every synchronizing pair t^A,t^B is merged into an internal event of C represented by a new event label $\tau t^A t^B$ (this is an enhancement to the internal event idea in CSP [5]).

(2) A resulting trace is constructed by interleaving of event labels from s^A and s^B; every occurrence of a synchronizing pair t^A, t^B such that t^A is directly followed by t^B or t^B is directly followed by t^A is replaced with $\tau e^A e^B$ according to the rule (1).

(3) An event label $t^A \in Sync_\mu(A,B)$ (as well as $t^B \in Sync_\mu(B,A)$) may not occur in a resulting trace and may only be processed via the rule (2). Traces where this condition would be violated are not included in the resulting behavior.

6.4.4. What is it Good for: Consistency Reasoning

The issues in consistency reasoning have been already articulated in Sect 2.x. Having refined the generic model Generic UC View into the extended models Basic UC View and Trace-based UC View, we rephrase the consistency issues with these extensions.

To reason on consistency of a hierarchical system specified via use cases, the following issues should be addressed:

(a) Does the composed behavior of the entities A_1, A_2, ..., A_n (forming a scope S) as specified by the use case expressions e_μ^{A1}, e_μ^{A2}, ..., e_μ^{An} (determining assembled behaviors of these entities) comply with the behavior specified for S by e_μ^S?

(b) Does the assembled behavior as specified by e_μ^A really reflect the desired behavior of A? As this is hard to address directly, we will consider equivalence checking – in general reasoning on whether two use case expressions e_μ and e_μ' specify the same behavior, i.e. whether $Com_\mu(e_\mu) = Com_\mu(e_\mu')$.

(c) Does an entity A_i communicate with its neighboring entity A_j (in the same scope S) in the way A_j expects (and vice versa), i.e., are the assembled behaviors e_μ^{Ai} and e_μ^{Aj} of A_i and A_j "compatible"?

Note that all these issues require obtaining assembled behavior of the entities involved, and, moreover, (a) requires obtaining the composed behavior of A_1,..,A_n. As an aside, we have defined in [70, 1] consistency relations for trace-based scenarios (by the concepts of language conformance and consent operator) addressing the issues (a) – (c). Moreover, a tool exists to evaluate the relations in an automatized way.

6.5. Analyzing UML 2.0: Assembled and Composed Behavior

In this section, we analyze how UML 2.0 supports assembled and composed behavior. By convention, we use *underlined italics* to refer to the abstractions defined by Basic UC View and ***slanted bold sans-serif font*** for UML metamodel elements (such as meta-classes, packages, associations). Figure 6.3 shows the parts of the UML metamodel related to use cases. Note that this figure is synthesized from several diagrams in the UML 2.0 specification [61].

In UML 2.0, an *entity* A is interpreted by a ***Classifier***. The ***UseCase*** metaclass interprets the concept of a *use case* $UC_\mu^A{}_j$; the ***Classifier*** associated via the ***subject*** association is the *SuD* (A) of the use case. A ***UseCase*** is associated with a ***Behavior*** specification via the ***ownedBehavior*** association (inherited from

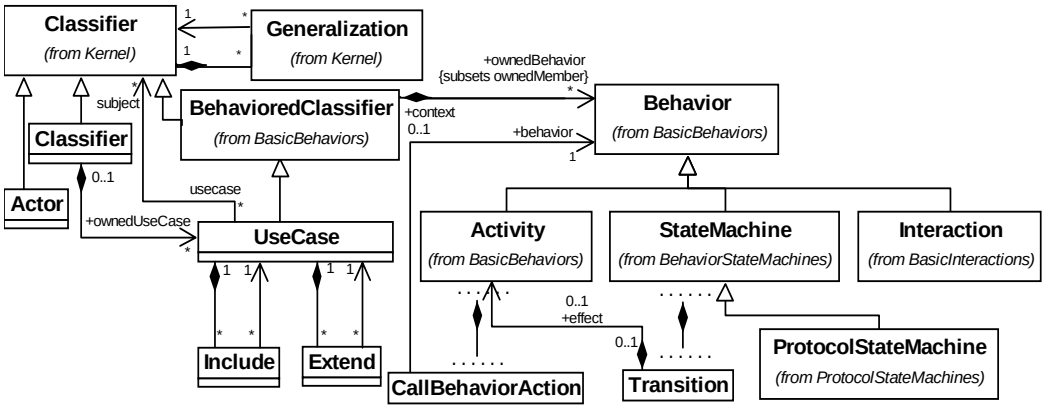


Figure 6.3: Relevant parts of the UML 2.0 Superstructure metamodel

BehavioredClassifier); the associated behavior may be an instance of one of the behavior specification mechanisms predefined in UML 2.0: ***Activity***, ***Interaction***, ***ProtocolStateMachine***, and ***StateMachine***. Therefore, in terms of Sect. 6.4, there are four concrete use case specification mechanism defined by UML 2.0.

For each of them, we below provide a brief characteristic and show: (i) how *Scenarios* are interpreted and whether a scenario is a trace, (ii) how and whether *assembled behavior*, (iii) *representative use case*, and (iv) *composed behavior* can be obtained, and (v) whether consistency reasoning is possible.

6.5.1. Interaction

In the ***Interaction*** specification mechanism (denoted Int), behavior is defined in terms of ***Messages*** sent among collaborating ***Classifier***s, resulting into partially-ordered sets of event occurrences (basically, ***Interactions*** are an enhancement of sequence diagrams from UML 1.5). A communicating ***Classifier*** is represented by a ***Lifeline*** (e.g., Clerk in Fig. 6.4). A ***Message*** (e.g., `lookupTicket`) connects two ***MessageEnds***. A ***MessageEnd*** may be attached to a ***Lifeline***, capturing an ***EventOccurrence*** – the message being sent or received (e.g., both ends of `lookupTicket`). Alternatively, a MessageEnd may be attached to the boundary of the Interaction, forming a ***Gate*** (`validatePayment` in Fig. 6.4).

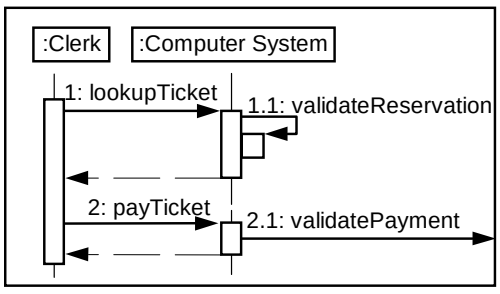


Figure 6.4: A use case of Computer System
captured as an ***Interaction***

Scenarios: Scenario is a trace – a sequence of event occurrences. More formally, behavior of an ***Interaction*** is defined by a pair [P,I], where P is the set of valid traces and I is the set of invalid traces. The set of event labels corresponding to operations associated with ***MessageEnd***s, together with the event kind (*send* or *receive*), form the domain of event labels Act_{Int}. Further, $Scenarios_{Int} = Act_{Int}^*$ and $Com_{Int}(UC_{Int}{}^{A}_{j}) = P$.

Assembled behavior: To define semantics of use case expressions, each operator from $\underline{OP}$ (used in Int) has to be associated with an operation θop. As scenarios are traces, for each of the operators (+ ; * |) such an operation is defined upon languages (sets of traces): union, concatenation, repetition and parallel composition (based on interleaving). Thus, $OP_{Int} = \underline{OP}$.

Representative use case: ***Interactions*** feature the construct ***CombinedFragment*** to combine ***Interactions*** together and ***InteractionOccurrence*** to refer to (include) an existing interaction. The semantics of ***CombinedFragment*** is defined in terms of the sets P and I, based upon the traces of its operands and an internal special operator. We define functions *Alt*, *Seq*, *Par,* and *Rep*, constructing a new ***Interaction*** as an ***CombinedFragment*** employing the operator ***alternative***, ***strict sequencing***, ***parallel execution***, and ***loop*** respectively. Each operand of the ***CombinedFragment*** is an ***InteractionOccurrence*** referring to the respective parameter of the function; these functions provide an implementation of all the operations in OP_{Int}. Thus, ***Interactions*** permit to construct a representative of all use case expressions.

Composed behavior: We interpret the set of event occurrences on ***Message-End***s that are a ***Gate*** as $Msg_{Int}(A)$. Further, we define $Pair_{Int}{}^{A,B}(t^A,t^B)$ iff (t^A=<*send*, *m*> ∧ t^B=<*receive*, *m*>) ∨ (t^A=<*receive*, *m*> ∧ t^B=<*send*, *m*>). As scenarios are traces, and $Pair_{Int}{}^{A,B}(t^A,t^B)$ is defined, composed behavior is interpreted in ***Interactions***.

Consistency reasoning: It is not explicitly addressed in ***Interactions***. However, as the semantics of ***Interactions*** is defined via traces, it is possible to use the relation of language conformance and the consent operator for consistency reasoning (Sect. 6.4.4).

6.5.2. Activity

The ***Activities*** specification mechanism (denoted Av) employs graphs similar to Petri Nets (PN). Briefly, a transition (PN) is a ***ControlNode***, a place (PN) is an ***ExecutableNode***, however, contrary to PN, the graph is not bipartite in general. An important subclass of ***ExecutableNode*** is ***Action***. Roughly speaking, an ***Activity*** specifies behavior in terms of passing tokens (control and data tokens) through the graph. In Fig. 6.5, `cs.lookupTicket` is a ***CallBehaviorAction*** (special case of ***Action***); below, a ***ForkNode*** (specialization of ***ControlNode***) creates two concurrent branches. Further, `cs.reportTicketPrice` is an ***AcceptEventAction*** (also special case of ***Action***) and the ***DecisionNode*** branching into alternative branches guarded by constraints `paymentDeclined` and `paymentProvided` is another specialization of ***ControlNode***.

Scenarios: An execution of an ***Activity*** generates a scenario, capturing history of the ***Actions*** executed. However, there is no assumption of atomicity of ***Actions*** (***Actions*** can overlap; an action “takes place over a period of time” [61, p.265]). Thus, an execution cannot be interpreted as a trace; UML 2.0 does not define any further details on $Scenarios_{Av}$.

Assembled behavior: As the semantics of parallelism and interleaving/overlapping of ***Actions*** is not clearly defined in ***Activities***, it is not possible to find an interpretation of "|". On the other hand, operators "+", ";" and "*" are each associated with an operation θop; these operations are defined via concatenation of scenarios. Therefore, OP_{Av} = { + , ; , * }.

Representative use case: In Activities, there is no construct to include another Activity via reference; therefore, the only way to implement operations from OP_{Av} with native operations is to construct a new graph actually including the graphs being combined. Note that ***CallBehaviorAction*** cannot be used as an "include" mechanism, because the semantics of this action is to call a new behavior (in a new execution context); the semantics of this action (in the definition of $Scenarios_{Av}$) is just the call of the behavior, and not the behavior being called itself.

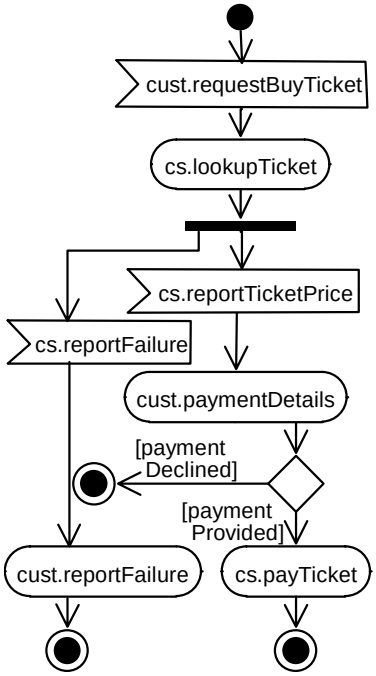


Figure 6.5: A use case of Clerk captured as an ***Activity***

Activity graphs can be combined together with ***ControlNodes***, however, only "+" can be interpreted this way, via ***DecisionNode***. As for interpretation of ";", it inherently needs to properly identify the end of execution of an ***Activity***. However, when an ***Activity*** creates multiple control tokens, this becomes a cumbersome task with too many special cases to handle (this should be a subject of further research). Therefore we conclude that it is not practically possible to implement *sequencing* (and neither *repetition)* via native operations and that constructing a representative use case is possible only in the case when the only operator used in the use case expression is "+".

Composed behavior: As $Scenarios_{Av}$ are not traces, it is not possible to interpret composed behavior in ***Activities***. Moreover, when considering whether it would be possible to establish a $Pair_{Av}^{A,B}$ relation, we see that requests are generated by an ***InvocationAction*** (or a subclass, e.g., ***CallBehaviorAction***), and received by an ***AcceptEventAction*** (or a subclass). However, these two class hierarchies have

different structure and it would not be possible to establish a $Pair_{Av}^{A,B}$ relation, even if $Scenarios_{Av}$ were traces.

Consistency reasoning: As $Scenarios_{Av}$ cannot be interpreted as traces, consistency reasoning is not possible.

6.5.3. State Machine

The ***StateMachines*** specification mechanism (denoted SM) is based on Harel state-charts [28], which enrich the automaton abstraction with composite states and their orthogonal regions. In principle, a transition from a composite state applies to all its substates; orthogonal regions model concurrently executing sub-machines. Harel state-charts define semantics via (i) event triggered transitions, and (ii) via atomic *actions* associated with entering or leaving a state, and with a transition. Time-durable *activities* are supported, but kept outside of the model. Special actions *start(X)* and *stop(X)* are used to start and stop an activity *X*. Thus, an execution of a Harel state-chart can be interpreted as a trace formed as a sequence of the action names. Also, accepting an event X may be captured as a special action *accepted(X)*.

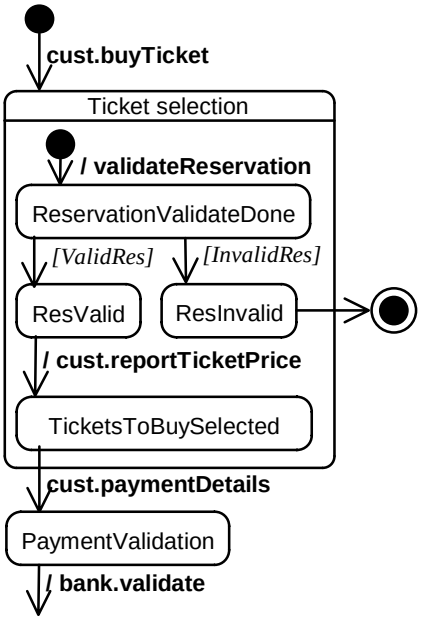


Figure 6.6: Fragment of a Computer System use case captured as a State Machine

On the contrary, UML 2.0 ***StateMachine*** replaces Harel's atomic action by the non-atomic ***Activity*** (Sect. 6.5.2). In addition, a ***State*** may also specify a ***doActivity***, which starts after entering the ***State*** (after completing the ***entry Activity***) and "is aborted prior to its completion" if the state is exited before the ***doActivity*** completes. However, UML 2.0 does not specify the exact semantics of *aborting* a ***doActivity***, neither the semantics of aborting the non-atomic ***Action***

possibly executed within a ***doActivity***. Because of this unclear semantics, it is hard to employ the Harel's start(X)/stop(X) trick to wrap a non-atomic ***Action***.

The State Machine demonstrated in Fig. 6.6 starts by receiving the event `cust.buyTicket`; the activity `validateReservation` is an internal action, while `cust.reportTicketPrice` invokes an operation. Guards are specified in square brackets.

Scenarios: As the ***Activities*** are non-atomic and the issue of aborting a ***doActivity*** is not resolved, it is not possible to interpret execution of a ***StateMachine*** as a trace in general. The concept of a scenario is not explicitly captured, we thus only implicitly assume the behavior of a ***StateMachine*** reflected as subset of $Scenarios_{SM}$, capturing histories of execution of ***StateMachines*** (***States*** visited, events processed, ***Transitions*** fired, and ***Activities*** executed).

Assembled behavior: The way we introduced $Scenarios_{SM}$ permits to define (based on concatenation) an operation θop to be associated with the operator "+", ";" and "*" respective. Although ***StateMachines*** explicitly consider parallelism (with orthogonal regions), parallel execution is limited by the *run-to-completion* semantics: an event t_2 may be accepted only after a running transition triggered by accepting an event t_1 completes (i.e., all the ***Activities*** associated with processing t_1 complete). Even though, as a special case, the transitions triggered by the same event execute in parallel, orthogonal regions do not interpret "|" in general and thus OP_{SM} = { + , ; , *}.

Representative use case: StateMachines permit to include (by reference) another StateMachine via a submachine state. To implement the operations in OP_{SM} ("+", ";" and "*"), we define functions *Alt*, *Seq*, and *Rep*, constructing a new ***StateMachine***. This ***StateMachine*** contains submachine states referring to the respective parameters of the function; the submachine states are joined via a ***junction PseudoState*** (Alt), via a ***Transition*** (Seq) and with a loop ***Transition*** (Rep) respectively; these functions implement all the operations of OP_{SM}.

Composed behavior: In ***StateMachines***, scenarios are not traces and thus, it is not possible to interpret composed behavior. Moreover, reception of events is captured in ***triggers*** associated with ***Transitions***, while responses are specified in the associated ***Activities***. Thus, even if $Scenarios_{SM}$ were traces, it would not be feasible neither to establish a $Pair_{SM}^{A,B}$ relation, nor to acquire composed behavior.

Consistency reasoning: As scenarios cannot be interpreted as traces (and moreover, as composed behavior is not interpreted), consistency reasoning is not possible.

6.5.4. Protocol State Machines

The ***ProtocolStateMachine*** specification mechanism (denoted PSM) is a specialization of ***StateMachine*** and is used to specify the correct order of sequencing of ***Operation*** calls on a ***Classifier***, typically an ***Interface*** (special view on an ***Classifier***). Unlike the general ***StateMachine***, PSMs are not permitted to associate ***Activities*** neither with a ***State*** (such as ***doActivitity***) nor with a ***Transition*** (the ***effect Activity***). A key point is that an ***Operation*** is specified in the ***trigger*** of the ***Transition*** (typically, the ***trigger*** will be a ***CallTrigger***, although other types of events are permitted as well). Thus, depending on whether the ***Interface*** is used as a ***provided*** or ***required Interface*** of a ***Port***, a PSM specifies either ***Operation*** call events consumed or emitted by the ***Interface***. However, as no

"reaction" events are specified, a single PSM can specify only one "direction of communication".

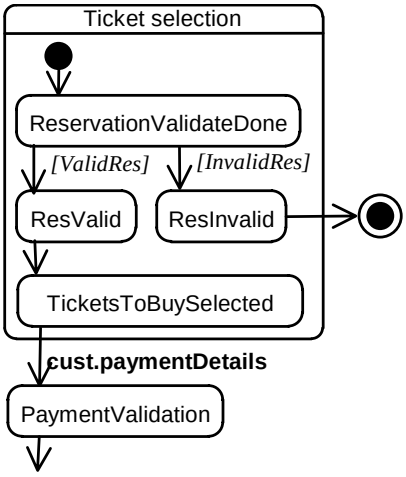


Figure 6.7: Fragment of a use case of Computer System captured as a Protocol State Machine

Figure 6.7 demonstrates a PSM defining the order of operation calls to be received by the Computer System while performing the behavior specified by the ***StateMachine*** in Fig. 6.6. The PSM has been acquired by capturing only the events processed, omitting all the activities.

Scenarios: A PSM captures the order of invocations of ***Operations*** specified as the ***trigger*** of its ***Transitions***. A PSM transition corresponds to the whole duration of the method invocation, and, same as in ***StateMachines***, the *run-to-completion* semantics requires that no event may be accepted until the processing of the previous event completes. Consequently, ***Operation*** invocations cannot interleave; only after an operation call completes another call may be started.

Even though UML2.0 does not introduce traces for PSMs explicitly, it is natural to interpret them as a sequence of ***Operation*** invocations; due to the non-atomicity of invocations, we represent an operation invocation by two atomic events, *request* and *response*; these events form the domain Act_{PSM}. Then, $Scenarios_{PSM} = Act_{PSM}^*$.

Assembled behavior: As a trace model is defined for PSMs, semantics of all the operators (+ ; * |) is defined by associating them with the same operations upon languages as in Sect. 6.5.1 (Activities) and thus, $OP_{PSM} = OP$. Note that "|" may create traces that cannot be generated by a PSM (also see below).

Representative use case: The operations "+", ";" and "*" can be interpreted via functions *Alt*, *Seq*, and *Rep*, defined in the same way as defined for StateMachines. However, as a ***Transition*** is non-atomic (represented in a trace by two events request and response), "|" may result into traces where other events interleave with the pair of the related request and response events. Due to the *run-to-completion* semantics of PoSMs, such a behavior cannot be represented with a PSM. Therefore "|" cannot be implemented with native operations and a representative use case can only be acquired when the operations are limited to "+", ";" and "*".

Composed behavior: As noted above, PSMs are intended to specify behavior of an ***Interface***, and a PSM specifies only the ***Operation*** call events but not the

"reaction events"; although $Msg_{PSM}(A)$ may be interpreted as the set of all events considered by a PSM, it is not possible to establish $Pair_{PSM}^{A,B}$. At least the former argument implies that *composed behavior* cannot be interpreted in ***ProtocolStateMachine***s.

Consistency reasoning: The support of traces in PSMs provides a solid basis for defining a consistency relation. This issue is explicitly considered in UML 2.0; a ***ProtocolConformance*** relation may be established between a PSM and a ***StateMachine*** (or between two PSMs). However, the semantics of the relation is not clearly defined.

6.6. Use Cases in UML 2.0: Additional Issues

In the previous section, we have discussed the behavior specification mechanisms that can be used to specify the behavior of a use case. In this section, we discuss additional issues related to use cases in UML 2.0 we identified in our analysis; we focus directly on the use case framework, i.e., on the metaclass ***UseCase*** and additional metaclasses declared in the package ***UseCases***.

6.6.1. Association between *UseCase* and *Behavior*

A ***UseCase*** is associated with a ***Behavior*** specification via the ***ownedBehavior*** association inherited from ***BehavioredClassifier***. The association is a composition with multiplicity 0..* on the target end (Behavior). The lower bound "0" permits a use case to have no ***Behavior*** associated, i.e., the behavior is specified outside the UML model. In such a situation no reasoning about the ***UseCase*** behavior is possible. The upper bound "*" permits a ***UseCase*** to be associated with more than one ***Behavior*** specification. For such a situation, it is not defined what would be the behavior of the UseCase. Although several implicit explanations may be identified, such as defining the behavior as the union of the partial behaviors, or considering the behaviors to be alternative representation of the same behavior, possibly employing different specification mechanisms, the UML 2.0 specification does not address this issue.

We may assume that this issues arises from the fact that ***UseCase*** only implicitly inherits the ***ownedBehavior*** association from ***BehavioredClassifier***; an explicit redefinition of this association in the ***UseCase*** metaclass would have the opportunity to elaborate the semantics of the multiplicity of the association.

Also note that the composition is exclusive (fig. 6.3) and thus, ***Behavior*** specifications cannot be reused among ***UseCase***s and a ***UseCase*** cannot associate as ***ownedBehavior*** a ***Behavior*** specification owned by its ***subject***.

6.6.2. Use Case Model not Defined

The concept of a *use case model*, which is either explicitly or implicitly considered by most use case modeling approaches, and which was included in UML 1.4/1.5 (***Model*** stereotyped `<<useCaseModel>>`), is omitted in UML 2.0. The only way to obtain the use case model of A is by implicitly including all use cases of A, i.e., all ***UseCases*** where A is the ***subject***. Note that this implicit solution does not permit multiple (alternate) models of A (UM^A and 'UM^A).

6.6.3. Subject of a UseCase: Multiplicity

A ***UseCase*** is associated with the entity it describes (a ***Classifier***, corresponding to *SuD*) via the ***subject*** association. The multiplicity of the association is 0..*, permitting a ***UseCase*** to be associated with more than one ***subject***, as well as with no ***subject***. However, the *use case* concept is tied with describing the behavior of a single *entity* (*SuD*); using a single ***UseCase*** for specifying multiple ***Classifiers*** violates these rules. When there is a need to reuse a single ***Behavior*** specification for multiple ***Classifiers***, multiple ***UseCases*** associated with the same ***Behavior*** should be used instead. As for the lower bound of the association's multiplicity ("0"), we find no interpretation for a ***UseCase*** not associated with an *entity* (***Classifier***).

6.6.4. Subject and Relations: Conflicting Constraint

A constraint imposed on the ***UseCase*** metaclass requires that *"UseCases can not have Associations to UseCases specifying the same subject"*. The constraint is not specified formally in the specification, the quoted sentence is the only source available. In our understanding of the constraint, *"Associations"* refers to the ***Include*** and ***Extend*** relations ***UseCase*** may be associated with. As the association between ***UseCase*** and its ***subject*** is *n* to *m* (* on both ends), the constraint requires that for UseCases UC_j and UC_k if ***Classifier*** *C* exists such that *C* is ***subject*** of both UC_j and UC_k, there is no ***Include*** or ***Extend*** relation *R* between UC_j and UC_k. (Here, we intentionally omitted the *SuD* reference from the use case identifier).

However, in use case practice, the *include* and *extend* relations are typically (almost exclusively) used among use cases describing the same subject; e.g., the *summary use case* typically uses the include relation this way.

The UML specification argues that *"Two use cases specifying the same subject cannot be associated since each of them individually describes a complete usage of the subject."* However, in our opinion, even a use case describing "complete usage" can be included in another use case, where additional behavior (preceding or following the use case's behavior) may be specified (e.g., in a summary use case).

Note that although the constraint prohibits directly relating use cases specifying the same subject, it does not restrict an indirect relation (either Include or Extend) via an intermediary use case not associated with the particular subject.

6.6.5. Use Case Relations not Reflecting Behavior Relations

The constructs available in the UML 2.0 behavior specification mechanisms permit a ***Behavior*** to invoke another ***Behavior***. Such a relation occurring between behavior specifications of different ***UseCase*** of the same ***Classifier*** is a typical realization of the concept of an *include* relation. Although such relations among ***Behavior***s are clearly captured in their UML specifications, and although the ***Include*** relation can capture the relation also among ***UseCase***s, there is no requirement to do so, i.e., to reflect the *invoke* relation among ***Behavior***s by an ***Include*** relation among the ***UseCase***s specified by these ***Behavior***s.

In a similar way, although the generalization relation available for ***StateMachines*** interprets the requirements of the ***Extend*** relation, there is no requirement to reflect

generalization between ***StateMachine***s by an ***Extend*** relation between the owning ***UseCase***s.

Note that this issue has been also identified in the OMG Issue No: 6445 processed by the UML 2.0 Finalization Task Force (FTF) [63].

6.6.6. Initiation of Communication

On a technical note, we have also discovered that UML 2.0 requires that the functionality specified by a use case *"is initiated by an actor"* (pp. 468, 10.3, UseCase). To symmetrically model communication between two entities (i.e., to create a use case model for each entity, where the entity is the ***subject*** (*SuD*) and the other is an ***Actor***), it is essential that a use case may be also initiated by its ***subject***.

6.6.7. Issues: Summary

In this section (Sect. 6.6) we have identified a number of issues related to how the *use case* concept is captured in UML 2.0. These issues are related to the issues identified by Isoda [32] in UML 1.5, which will be discussed in Sect. 8.3.

6.7. Evaluation

We claim that assembled behavior, representative use case, and composed behavior are important concepts which should be reflected in software design tools to make the use case idea really work. This claim is also inspired by the practitioners' concept of summary use case (e.g. [13]). Contrary to UML, we do not consider in our Basic UC View the ***Include*** and ***Extend*** use case relations as we focus on assembled and composed behaviors where no relations upon use cases are needed. Also some behavior specification mechanism feature a construct "include" resp. "extends"; however non of them can be interpreted as a binary operation $U_{\mu}^{A} \times U_{\mu}^{A} \rightarrow U_{\mu}^{A}$ (nor, in general, as an operation upon specifications only). The bottom line is that these constructs are not significant in our approach as the behavior of an included use case seamlessly becomes a part of behavior of the including use case. Thus, we "skip" use cases included in another use case, considering "complete" use cases only.

The way we modify Generic UC View in this paper allows to analyze the behavior specification mechanisms in the emerging UML 2.0. We conclude that only ***Interactions*** define trace-based scenarios and support obtaining assembled behavior, representative, use case as well as composed behavior. On the contrary, behavior of ***Activities*** and ***StateMachines*** cannot be interpreted via traces. Assembled behavior can be obtained only for a limited set of assembly operators ($OP_{AV} = OP_{SM} = \{+ ; *\}$). Moreover, while for ***StateMachines*** the whole OP_{SM} can be implemented via native operations, for ***Activities*** only "+" is the case. Only for expressions featuring solely these (natively implemented) operators a representative use case can be constructed. Behavior of ***ProtocolStateMachines*** (PSM) can be captured as traces, however, PSMs are not designed to specify the behavior of an entity, but only of a single ***Interface*** of a ***Port***. Thus, PSMs do not provide an interpretation of composed behavior either.

To justify that the idea of assembled behavior, representative use case, composed behavior, and consistency reasoning is sound, we refer the reader to Pro-cases [67, 69], based on behavior protocols [70] as a model of Generic UC View (they also fit into the Trace-Based UC View). Pro-cases (PC) feature trace-based scenarios and thus, all behavior assembly operations are defined on traces (OP_{PC} = $\underline{OP}$); moreover, native operations exist to implement the whole OP_{PC}. Composed behavior is interpreted and a decidable consistency relation defined. A tool exists to verify this relation.

6.8. Conclusion (Chapter Summary)

We demonstrated Basic UC View, a simple formal model capturing key abstractions in use case specifications. We employed it and its specialization Trace-Based UC View to analyze the four behavior specification mechanisms available in UML 2.0. As a result we showed that none of these mechanisms explicitly addresses assembled behavior, representative use case, nor composed behavior. However, we draw a way to interpret these concepts in ***Interactions***.

Chapter 7

Port State Machines

posm

Protocol State Machines (PSM) in UML 2.0 [61] describe valid sequences of operation calls. To support modeling components, UML 2.0 introduces a *Port* associated with a set of provided and required interfaces. Unfortunately, a PSM is applicable only to a single interface, either a provided or required one; moreover, nested calls cannot be modeled with a PSM. Furthermore, the definition of *protocol conformance* is not precise enough to permit reasoning on the relation in general; thus reasoning on consistency in component composition is not possible with PSMs.

In this chapter, we propose the Port State Machine (PoSM) to model the communication on a Port. We base PoSM on UML 2.0 Protocol State Machines [61] and employ the formal model of behavior protocols [70]. Building on our experience with behavior protocols, we model an operation call as two atomic events *request* and *response*, permitting PoSM to capture the interleaving and nesting of operation calls on provided and required interfaces of the Port. The trace semantics of PoSM yields a regular language. We apply the compliance relation of behavior protocols to PoSMs, allowing us to reason on behavior compliance of components in software architectures; the existing verifier tool can be applied to PoSMs.

7.1. Introduction

7.1.1. UML 2.0: State Machines and Protocol State Machines

> In this section, we introduce some of the features of UML 2.0. While the behavior framework of UML 2.0 has already been described in Sect. 6.1, the description below is an integral part of this chapter, as it focuses on the aspects that motivate us to introduce the Port State Machines.

The Unified Modeling Language (UML) [60] features ***StateMachines*** based on the widely recognized State-chart notation [28]; the execution of a State Machine can be observed in terms of *events* accepted and *actions* executed (potentially overlapping). The upcoming new version of the standard, UML 2.0 [61], introduces a specialization of ***State Machine***, the ***Protocol State Machine*** (PSM), which can be used to model the ordering of operation calls on a ***Classifier*** (typically an ***Interface***). Moreover, UML 2.0 introduces the concepts ***StructuredClassifier*** and ***EncapsulatedClassifier***, providing support for modeling internal structure and featuring Ports; a ***Port*** is associated with a set of ***provided*** and ***required*** interfaces. Based on these concepts, a ***Component*** may be captured in a UML model, employing a possibly hierarchical component model; the external communication of the component is encapsulated in the component's ***Ports***.

In component-based software engineering, a basis for reasoning on “compatibility” of software components is highly desirable in order to validate software architectures and define substitutability of components.

UML explicitly considers *"conformance"* of PSMs; however, the role of conformance is limited to explicitly declaring, via the ***ProtocolConformance*** model element, that a *specific* StateMachine (possibly a PSM) conforms to a *general* PSM. Note that UML defines the semantics of protocol conformance only partially (based on structural equivalence and matching guards on transitions); it is not clear under which circumstances protocol conformance may be declared and thus, it is not feasible to automatically decide on protocol conformance.

UML employs the protocol conformance in the ***Components*** framework, requiring *realization* of a ***Component*** (possibly a StateMachine specifying the component) to be conforming with specifications of all its ***Interfaces***. Moreover, when a required interface I^R is connected to a provided interface I^P, the PSM of I^R must be conforming to the PSM of I^P. However, with no exact definition of protocol conformance, validating component architectures is not feasible.

7.1.2. Motivations

Although the State Machines in UML permit modelers to clearly communicate ideas to each other, they are not suitable to be used as the basis for defining "compatibility" of components. The observable behavior of a component is typically captured as communication on its *provided* and *required* interfaces [9, 17, 26, 66]. However, in UML State Machines, significantly different mechanisms are employed to specify events received and sent. Events received (in case of a component corresponding to operations on the provided interfaces), are captured as *triggers* associated with transitions of the state machine. A State Machine uses ***Activities*** to specify its responses to events received (i.e., events sent and internal actions). An ***Activity*** (a Petri-net like abstraction in principle) consists of ***Actions***, some of these actions correspond to sending events. However, the spectrum of actions is rather huge and it is not possible to establish a one-to-one correspondence between the triggers and actions related to a communication; thus, it is not possible to derive the behavior resulting from the composition of communicating components (exchanging events) specified with State Machines.

A Protocol State Machine (further PSM), a refinement of the (generic) *behavioral* State Machine, imposes a restriction on its transitions, requiring that no ***Activities*** are associated with the execution of the PSM. Consequently, only one "direction of communication" can be captured with a PSM. The communication is specified independently of the direction of communication, only the way an interface described by a PSM is used in a Port determines whether the events captured by the PSM are received (for provided) or sent (for a required interface); a PSM cannot describe the interplay of events on the provided and required interfaces.

UML State Machines employ the *run-to-completion* semantics, i.e., only after a transition of a State Machine completes can another event be processed. Thus, while executing a method (modeled, e.g., as the effect activity of the transition), no other event may be processed by the State Machine, i.e., no other method call may be accepted. Thus, State Machines cannot capture interleaving of calls (several incoming calls processed at the same time), and neither nested calls (e.g., a call-back or statically limited recursion), nor they support (unlimited) recursion.

Surprisingly, the situation is no easier in PSMs – although no activity corresponding to the operation called is included in a PSM, a transition completes only after the method implementing the operation completes. Therefore, no call may be accepted before the call being processed completes and thus, the same

restrictions on call interleaving and nested calls apply to PSMs. Consequently, although a PSM specifies a sequence of operation calls, the communication on a Port, associated with provided and required interfaces described by PSMs, cannot be captured with *traces* for further behavioral reasoning, due to the non-atomicity of the events (operation-call) in the sequences described by the PSMs. Moreover, the descriptions cannot capture nesting and interleaving of calls on the Port, although this is a common pattern in component communication.

Considering the lack of well-defined semantics, establishing a rigorously defined compatibility relation upon the behavior of PSMs is not feasible. Among others, UML assumes a constraint language to be used for guards of transitions, but no constraint language is prescribed; OCL is provided only as one of the options. Moreover, even when assuming OCL to be the only constraint language permitted, such a relation would hardly be decidable for the following reasons: (i) OCL expressions may access the attributes of the classifier, i.e., an internal state with potentially unlimited state space and (ii) Events may be deferred and processed later, thus the automaton gets a stack (though no semantics is given for the order of retrieval; thus the event pool rather resembles a bag). Here, the consensus is that verification of compliance is feasible only on regular automata (or other abstractions with equivalent expressive power). In certain cases, the relation may be decidable for a context-free grammar / stack automaton; however, actually evaluating (computationally) such a relation is likely to be unfeasible in general. A compliance relation is typically defined on regular languages, e.g., a decidable relation is defined in [70]; the work on the consent operator [1] provides an alternative approach [2]. Note that the approach taken in [41, 42] also uses a subset of statecharts that can be converted to a finite LTS.

In case a trace model can be defined for the sequences of events described by a state machine (here, it is essential that the events are atomic), reasoning on compliance may be possible. When defining behavioral compliance, we see as important that (i) compliance is based only on the behavior described and not on the structure of the specification (ii) compliance is unambiguously defined (iii) deciding on compliance can be achieved in an automated way. Unfortunately, none of these is the case for ***ProtocolConformance*** defined in UML 2.0 (as discussed in Sect. 7.1.1).

Last but not least, we miss a layer of description between a PSM (focused on a single interface) and a behavioral State Machine specifying a component, i.e., a layer suitable for specifying communication on a ***Port*** (of a component).

Thus, the issues we identified are: (i) State Machines in UML do not capture interleaving of sent and received events. (ii) Composition of State Machines is not possible (iii) The form State Machines use does not permit establishing a decidable compliance relation. (iv) A specification mechanism is missing to capture the communication on a ***Port***.

7.1.3. Goals and Structure of the Chapter

In [70], our research group developed Behavior Protocols, modeling behavior of *agents* as traces of atomic events. Applied to the SOFA component model [66], behavior protocols capture the ordering of operation calls issued and handled by a SOFA component. Nesting of other events (possibly also operation calls) within an operation call is supported. Moreover, a decidable compliance relation is defined; a verifier tool [46, 77] for checking this relation is available. SOFA is a hierarchical

component model; a component (either *primitive* or *composed*) communicates with its neighboring components via a set of *provided* and *required* interfaces. The abstraction of a software component, as considered in SOFA, employs a set of features comparable to those available in UML 2.0 Components.

Considering the motivations discussed in Sect. 7.1.2, we propose *Port State Machine* (PoSM) with the following goals: (1) Provide a notation that allows to capture interleaving of events sent and received (by a Port of a Component) and support nested calls in such a way that the behavior can be captured with a trace model based on atomic events. (2) Moreover, a verifiable compliance relation should be defined for PoSMs.

This chapter is structured as follows: Sect. 7.2 introduces the Port State Machines (PoSMs), in Sect. 7.3, we show how composition verification can be achieved with PoSMs; a case study follows in Sect. 7.4. In Sect. 7.5, we evaluate PoSMs as an instance of Trace-based UC View. Section 7.6 evaluates the contribution; we summarize the chapter in Sect. 7.7. Related work in discussed in Chapter 8; we explore future work opportunities in Sect. 9.3.3.

7.1.4. Note on Conventions Used

In this chapter, PSM stands for Protocol State Machines (introduced by UML 2.0), while PoSM (at convenience pronounced "possum") stands for Port State Machines, proposed in this paper. A ***slanted bold sans-serif*** font is used to distinguish UML metamodel identifiers (names of packages, metaclasses, associations and attributes).

7.2. Port State Machines

We propose Port State Machines, building upon the UML 2.0 Protocol State Machines. To model operation calls (inherently non-atomic) with atomic events, PoSMs capture an operation call with two events, *request* (corresponding to start of the operation call) and *response* (completion of the operation call). Moreover, PoSMs explicitly distinguish between *sent* and *received* events. Here, an operation call handled on a provided interface is represented by a received request event and a sent response event; in a similar way, an operation call issued on a required interface is represented by a sent request event and a received response event. To hide such technical details from the modeler, PoSM notation defines convenient shortcuts. In Fig. 7.1, a Port State Machine explicitly specifies the request and response events, while in Fig. 7.2, the same behavior is described with the notation shortcuts (these will be described in Sect. 7.2.3).

7.2.1. PortStateMachine Metamodel

We define ***PortStateMachine*** as an extension of UML 2.0 ***ProtocolStateMachine***, employing the UML 2.0 extension mechanisms. In Fig. 7.1, we show the newly introduced metaclasses ***PortStateMachine*** and ***PortTransition***, as well as the related classes of the UML ***StateMachine*** specification to provide context for our extension.

PortStateMachine is defined as a subclass of ***ProtocolStateMachine***. Thus, a PoSM contains one or more regions; a ***Region*** contains vertexes and transitions. A ***Transition*** connects a ***source*** vertex to a ***target*** vertex. A ***Vertex*** may be a ***PseudoState*** or a ***State***. A ***PseudoState*** is a syntactic construct to model entry and exit points of regions.

A ***State*** may contain zero or more ***regions***. A ***State*** not containing any region is a *simple* state. ***FinalNode*** is a specialization of a ***State*** representing the completion of a region; a ***FinalNode*** may not contain any regions or have outgoing transitions.

A ***State*** containing one or more regions is a *composite* state, a syntactic construct to provide hierarchical grouping of states. When a simple state is active, all its containing composite states are active. In an active composite state, one of its substates is active.

A ***State*** containing more than one region is an *orthogonal state*. Orthogonal states model concurrent execution; in an active orthogonal state, a substate is active in each of its regions and a transition may be taken in any of the regions.

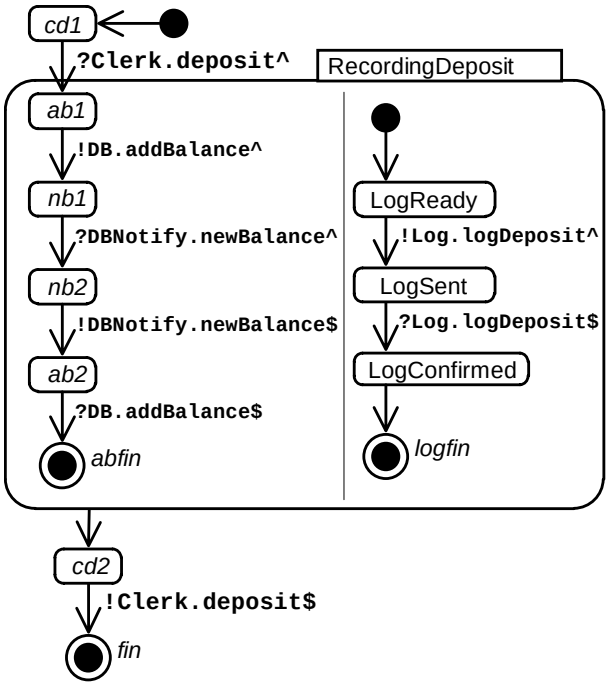


Figure 7.1: Port State Machine with explicit request and response transitions

Example 7.1 In Fig. 7.1, **`RecordingDeposit`** is an orthogonal state; the calls **`DB.addBalance`** and **`Log.logDeposit`** progress in its orthogonal regions independently.

A ***PortTransition*** represents a single atomic communication event. ***PortTransition*** is a subclass of ***ProtocolTransition*** (which in turn is a subclass of ***Transition***). A ***PortTransition*** must have exactly one ***trigger***; the ***trigger*** must be a ***CallTrigger*** and must refer to an ***Operation*** of an ***Interface*** of the ***Port*** associated with the PoSM. ***PortTransition*** introduces two additional attributes, both

of an enumerate type: ***OperationCallPart*** captures whether the transition represents the ***request*** or ***response*** part of the operation call. ***CommunicationDirection*** specifies whether the event is ***received*** or ***sent***, its value must be ***sent*** for a ***request*** on a ***required*** interface or a ***response*** on a ***provided*** interface, and ***received*** in the opposite cases (***response*** on ***required*** interface or ***request*** on a ***provided*** interface).

Example 7.2 In Fig. 7.1, transition from **`LogReady`** to **`LogSent`** denotes sending a request for operation **`Log.logDeposit`**, while the ongoing transition to **`LogConfirmed`** denotes receiving the response for this operation. (The notation will be described in detail in Sect. 7.2.3)

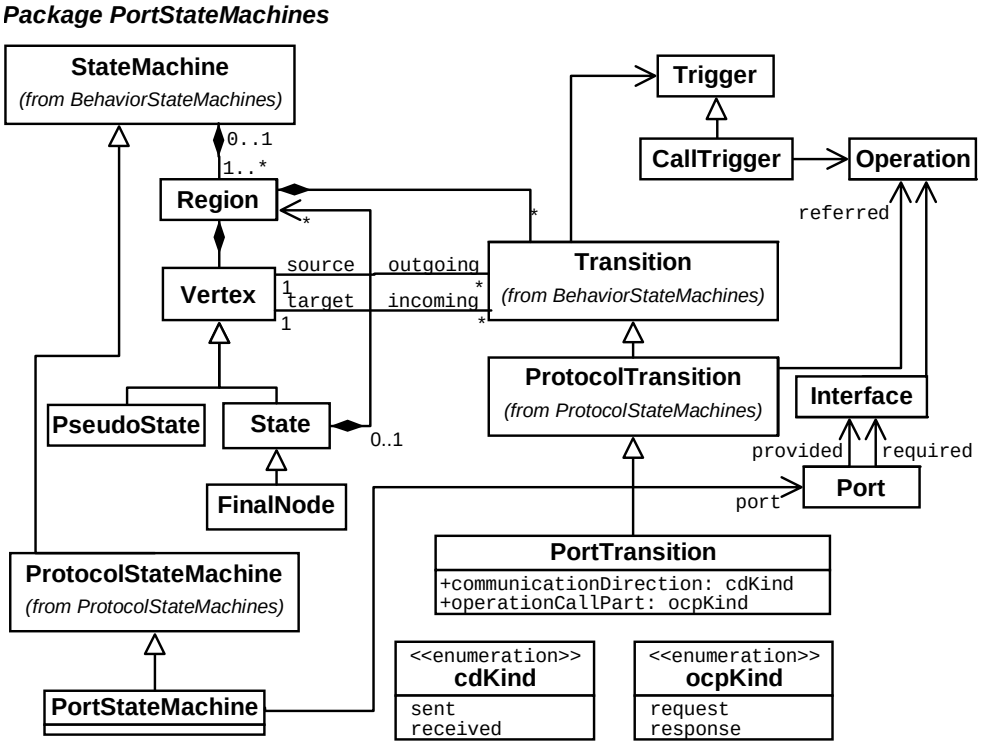


Figure 7.1: Port State Machines abstract syntax: definition of ***PortStateMachine*** and ***PortTransition***. The owning package name is shown only for selected metaclasses; of the others, metaclasses ***Region***, ***Vertex***, ***PseudoState***, ***State*** and ***FinalNode*** are owned by ***BehavioralMachines***, ***Trigger*** and ***CallTrigger*** by ***CommonBehaviors***, ***Operation*** by ***Kernel***, ***Interface*** by ***Interfaces*** and ***Port*** by ***CompositeStructures***.

With the goal to provide trace semantics and facilitate a compliance relation (as discussed in 7.1.2), PoSMs introduce the following additional restrictions on ***PortStateMachine*** instances and its contained elements:

(i) A transition in a ***PortStateMachine*** must be either a ***PortTransition*** or a ***ProtocolTransition***; a transition that is not a ***PortTransition*** may not specify any triggers, i.e., can only accept the *completion event*. A ***PortTransition*** may only originate in a ***State*** (but may target a ***PseudoStates***).

(ii) A transition in a ***PortStateMachine*** may not specify any constraints – its ***guard***, ***preCondition*** and ***postCondition*** associations must be empty.

(iii) The ***deferrableTrigger*** association of each ***State*** must be empty.
(iv) Only one transition is taken for a single occurrence of an event, even when multiple transitions in different regions of an orthogonal state specify the same trigger (opposite to the UML 2.0 orthogonal state semantics where all such transitions are taken simultaneously).
(v) The ***kind*** of a ***PseudoStates*** in a PoSM must be either ***initial*** or ***fork***. For the sake of simplicity of the PoSM definition, we omitted the other ***PseudoState*** kinds: ***choice*** and ***junction*** (not meaningful without guards), ***deepHistory*** and ***shallowHistory*** (complex semantics; can be replaced with increased state space), ***join*** (complex semantics, can be partially replaced with FinalNodes) and ***exit*** and ***terminate*** (we focus on complete traces).
(vi) A transition from ***PseudoState*** may only target a vertex recursively contained by the region containing the PseudoState. (I.e., may not cross state boundaries outwards).

The restrictions specified above together with the constraints initially specified by UML 2.0 [61] assure certain properties; we highlight here those that will be used later:

(i) A transition originating from a ***State*** may cross several boundaries of containing states outwards, then cross several boundaries of composite states inwards and finally targets a ***Vertex***,
(ii) A transition originating from a ***PseudoState*** is not a ***PortTransition*** but only a ***ProtocolTransition***. A transition from an ***initial PseudoState*** within a region *r* either targets a ***Vertex*** directly contained by *r* or a ***Vertex*** within a ***State*** contained by *r*. Only one transition may originate from an ***initial PseudoState***.
(iii) Given a ***fork PseudoState*** p_f contained in region *r* containing also a composite state *s*, multiple transitions may originate from p_f, each targeting a ***Vertex*** in a different region of *s*.

Example 7.3 The PoSM shown in Fig. 7.1, after receiving a request for operation **`Clerk.deposit`**, enters the orthogonal state **`RecordingDeposit`**. After both its regions complete, the PoSM eventually sends a response for the **`Clerk.deposit`** operation.

7.2.2. Trace Semantics of Port State Machines

We define the semantics of a PoSM P^A via the traces generated by P^A. We model the behavior as traces of *state events* and *communication events* forming the *communication language* and *execution language* of P^A.

Definition 7.4 Let *St* be the set of all states and let *Reg* be the set of all regions, directly or indirectly contained in a PoSM P^A. For a region $r \in Reg$, we denote *States(r)* the set of states directly contained by *r*. In the same vein, for a state $s \in St$, *Regions(s)* is the set of regions directly contained by *s*. *Regions(P^A)* is the set of top-level regions directly contained by P^A.

Definition 7.5 State s_i is a *substate* of s_j, if there is $r \in Regions(s_j)$ such that $s_i \in States(r)$. A state s_j *recursively contains* a state s_i if s_i is a substate of s_j, or there is a substate s_k of s_j such that s_k recursively contains s_i.

A region r_j recursively contains a state s_i, if $s_i \in States(r_j)$ or there is $s_j \in States(r_j)$ such that s_j recursively contains s_i.

Definition 7.6 Let *SL* be set of labels for states in *St* and *OL* be the set of labels for operations associated with transitions of P^A.
We define the domain of state events $SE = \{ entry, exit \} \times SL$ and the domain of communication events $CE = \{ sent, received \} \times OL \times \{ request, response \}$. The set *CE* is the domain of events for communication traces of P^A and the set $S = SE \cup CE$ is the domain of events for execution traces of P^A. Note that state events only capture entering or leaving a ***State***, but not a ***PseudoState***.

Definition 7.7 A *configuration c* of P^A is a subset of *St* for which both the following conditions hold:
(i) for each region $r \in Reg$, *c* contains at most one state $s \in States(r)$
(ii) if $s_i \in c$ and s_i is a substate of s_j, then $s_j \in c$.

A state s_i is *active* in *c* if $s_i \in c$. A region *r* is *active* in *c* if there is a state $s_i \in States(r)c$ such that $s_i \in c$.
A configuration *c* is *stable*, if each top-level region of P^A is active and for each state $s_i \in c$, all regions of s_i are active.

Definition 7.8 The *label* of a ***PortTransition*** *T* associated via its ***trigger*** with an operation *op* is the event $e = < cd_T, label_{op}, ocp_T > \in CE$, where cd_T and ocp_T are the communication direction and operation call part attributes of *T* and $label_{op} \in OL$ is the label for *op*. A transition not associated with an operation does not have a label.
In a configuration *c* (of P^A) containing a state s_i, P^A may take a transition *T* originating from s_i, iff at least one of the following conditions holds:
(i) *T* has no label, and either s_i has no regions, or the active state of all regions of s_i is a ***FinalNode***.
(ii) *T* is a ***PortTransition*** labeled with event $e \in CE$ and there is no state $s_j \in c$ such that (1) s_i recursively contains s_j and (2) a transition *U* also labeled *e* originates from s_j (in this case, we say that *U* has *higher priority* than *T*).

The innermost region recursively containing the source and target vertexes of *T* is the *least common ancestor* (LCA) of T, denoted $r_{lca,T}$. The LCA configuration $c_{lca,T}$ is obtained from c_1 by removing all states recursively contained in $r_{lca,T}$.

Example 7.9 For the PoSM shown in Fig. 7.1, { *RecordingDeposit, ab1, logfin* } is a stable configuration, in which $ab1 \mapsto ab2$ is the only legal transition and the left region of *RecordingDeposit* is the LCA.
Configuration { *RecordingDeposit, abfin, logfin* } is also a stable configuration of this PoSM, where the only legal transition is $RecordingDeposit \mapsto cd2$; here, the single topmost region of the PoSM is the LCA.

Definition 7.10 From $c_{lca,T}$, *T* determines the target stable configuration the following way:
(i) T is an *engaged* transition.
(ii) An engaged transition targeting a ***State*** *s* causes *s* to become *active*.
(iii) An engaged transition targeting a ***PseudoState*** *p* causes transitions outgoing from *p* to be *engaged*.

(iv) All containing states of a state that becomes active become active (if they are not active yet).

(v) For each composite state that becomes active, all regions become active. If an engaged transition T_i targets a vertex in a region r, r becomes active by T_i *explicitly* and T_i determines the active state of r. Otherwise, r becomes active *implicitly* and the transition originating from an ***initial PseudoState*** of r becomes engaged. If there is no such transition, the model is ill-formed. Eventually, after processing all engaged transitions and according to these rules, all regions that must become active have an active state selected, yielding a stable configuration c_2. By observation, $c_{lca,T} \subseteq c_1 \cap c_2$.

A single *run-to-completion step* of P^A from a stable configuration c_1 to a stable configuration c_2 initiated by a transition T is captured with a trace $t_{T,k}$, acquired as concatenation of parts $t_{T,exit}$, $t_{T,com}$ and $t_{T,entry}$. The first part $t_{T,exit}$ is a sequence of state exit events reflecting the transformation of c_1 to $c_{lca,T}$ via a sequence of valid configurations $c_{j,exit}$. Next, $t_{T,com}$ contains the label of T if T is a ***PortTransition***, or is a null sequence otherwise. Finally, $t_{T,entry}$ is a sequence of state entry events reflecting the transformation of $c_{lca,T}$ to c_2 via a sequence of valid configurations $c_{j,entry}$. Note that due to the loose ordering constraints on entry and exit events for orthogonal states, there may be multiple traces $t_{T,k}$ capturing the *run-to-completion step* from c_1 to c_2 via T.

The initial stable configuration $c_{init,PA}$ of P^A is determined by transitions from ***initial PseudoStates*** of top-level regions of P^A. Configuration $c_{fin,PA,k}$ is a final configuration of P^A if the active state of each top-level region of P^A is a ***FinalNode***.

We capture a single *run* of P^A with a trace t_{PA}, acquired as concatenation of parts $t_{PA,entry}$, $t_{PA,k}$ and $t_{PA,exit}$, where $t_{PA,entry}$ is the sequence of state entry events to reach the initial stable configuration $c_{init,PA}$ of P^A from the empty configuration, $t_{PA,k}$ is concatenation of a finite sequence of traces capturing a sequence of run-to-completion steps reaching a final configuration $c_{fin,PA,k}$ from $c_{init,PA}$, and $t_{PA,exit}$ is a sequence of the state exit events to reach the empty configuration from a $c_{fin,PA,k}$.

Definition 7.11 The set of all traces of all possible runs of P^A forms the *execution language* of P^A, denoted $LE(P^A)$. *Communication language* of P^A, denoted $LC(P^A)$ is the restriction of $LE(P^A)$ to the domain of communication events *CE*.

Example 7.12 The transition *RecordingDeposit* ↦ *cd2* of the PoSM shown in Fig. 7.1 may be captured with the following trace:

$$< exit_{abfin}, exit_{logfin}, exit_{RecordingDeposit}, entry_{cd2} >$$

This trace (which does not contain any communication event) may be followed in a run of the PoSM by a trace of the transition *cd2* ↦ *fin* (labeled with sending a response for Clerk.deposit; we use *?* to denote receive, *!* for send, ↑ for request and ↓ for response):

$$< exit_{cd2}, !Clerk.deposit\uparrow, entry_{abfin} >$$

The following example is a possible trace from the communication language of this PoSM:

< *?Clerk.deposit*↑, *!Log.logDeposit*↑, *!DB.addBalance*↑,
?DBNotify.newBalance↑, *!DBNotify.newBalance*↓, *?DB.addBalance*↓,
?Log.logDeposit↓, *!Clerk.deposit*↓ >

7.2.3. Notation

PoSMs introduce extensions to the UML state machine notation, with the goal to avoid an increase in complexity of the state machine diagrams, even when capturing the additional information required by PoSMs. In particular, the extensions permit to: (i) capture the additional attributes of a transition in its label, (ii) use implicit intermediate states and (iii) capture nested calls. The notation shortcuts are demonstrated in Fig. 7.2, concisely describing the same behavior as the PoSM in Fig. 7.1 (not employing the shortcuts).

The PoSM notation utilizes the notation of Behavior Protocols (BP) [71, 70],. There, the event token *?a* stands for receiving an event *a* and *!a* for sending an event *a*. A call of an operation *op* is captured with a pair of atomic events; in the event labels, the suffix ↑ denotes request and ↓ response. E.g., sequence *?op*↑***;***!op*↓ (here ***;*** is the operator for sequencing) models receiving call of the operation *op* as receiving a request for *op* and sending a response. In BP, the shortcuts *?op* and *!op* stand for sequences *?op*↑ ***;*** *!op*↓ and *!op*↑ ***;*** *?op*↓; shortcuts *?op{ Prot }* and *!op{ Prot }* stand for *?op*↑ ***;*** *Prot* ***;*** *!op*↓ and *!op*↑ ***;*** *Prot* ***;*** *?op*↓ respective.

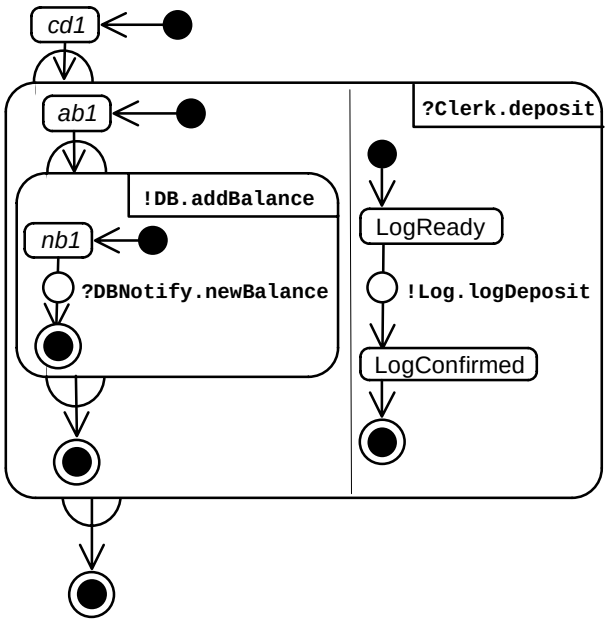


Figure 7.2: Port State Machine employing *call transition* and *call state* shortcuts

The notation for PoSMs employs these prefixes (*?/!*) and suffixes (↑/↓) in the event label to express the attributes of a ***PortTransition***. Due to the (possible) limitations of the character set available in UML (actually, no character set is specified), we represent ↑ with **^** and ↓ with **$** respectively. We demonstrate the notation and the shortcuts in Fig. 7.3.

Figure 7.3 (a) models receiving call of operation *a*, explicitly captured as two PortTransitions adjoined in an (explicit) intermediate state. Fig. 7.3 (b) employs a *call transition* as a shortcut to model the same sequence. A single call transition

represents two ***PortTransitions*** and the intermediate state; mnemonically, the circle on the call transition reminds of the implicit intermediate state. The single label of the call transition ("**?a**") determines the trigger of both the PortTransitions; the communication direction of the first (***request***) transition is equal to the symbol used in the label, while the communication direction of the second (***response***) transition is the opposite.

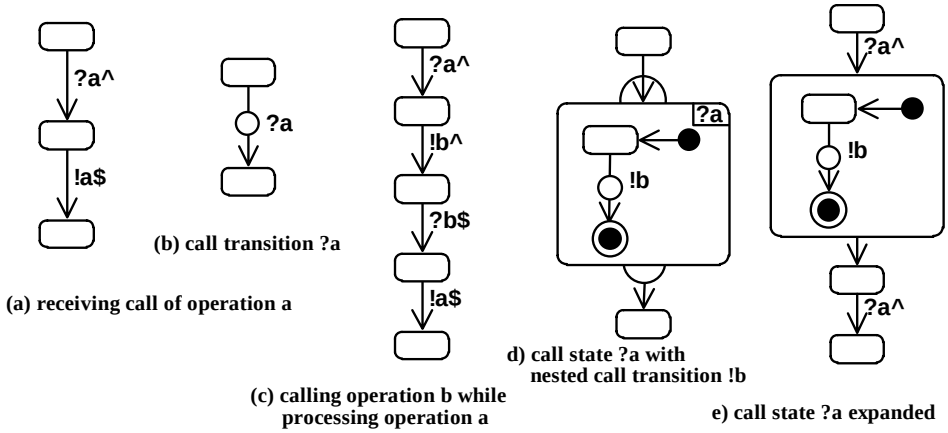


Figure 7.3: Port State Machines notation

PoSM notation also provides a syntactic construct to model nested calls. In Fig. 7.3 (c), operation *b* is called while operation *a* is being processed. The same sequence of events can be captured with a *call state*, as demonstrated in Fig. 7.3 (d). The call state construct represents a structure of transitions and states, the core of which is a composite state containing the behavior that occurs between the request and response.

In the same way as for a call transition, two ***PortTransitions*** (for request and response) are specified with only one occurrence of the operation call label. To preserve the general operation call semantics, a call state may only be entered with the request transition, may complete only after its internal behavior completes, and must complete with the response transition. Thus, a call state may have only one incoming and one outgoing transition. Moreover, to assure that the composite state may only exit with the completion event, the composite state exits with an unlabeled transition targeting an intermediate state, from which the response transition originates. Figure 7.3 (e) demonstrates the composite state, intermediate state and transitions represented by the call state shortcut in Fig. 7.3 (d).

The incoming transition of the call state must target the state itself, an ***initial PseudoState*** has to be used to specify where the region(s) of the call state start (a call state may have multiple regions). Syntactically, a call state employs the notation for a composite state and is distinguished with two semicircles attached to the top and bottom of the state; the operation call label is placed in the top-right corner.

Note that throughout this example, we used for brevity the symbols *a* and *b* to refer to an operation on an interface. Clearly, an identifier of the interface and an identifier of the operation are required to identify the operation unambiguously; in the other examples (e.g., figs. 7.1 and 7.2) the character "." (dot) is used to join these identifiers.

Also please note that we define the call state and call transition notation shortcuts by specifying how they expand into elements defined in the PoSM metamodel. We

consider this approach to be the most efficient with respect to readability of the paper, in particular of the trace semantics definition. Alternatively, we might define call state and call transition in the metamodel and either extend the semantics definition also to these elements, or to define a transformation of a model employing these constructs to a model based only on the already considered metamodel elements; both these approaches are feasible.

7.2.4. Properties of Communication Traces

In this section, we define the well-formedness property of communication traces and show its relation to the PoSM notation shortcuts; moreover, we also claim that the communication language of a PoSM is a regular language, we support this claim with a proof sketch.

Definition 7.13 A communication trace *t* is *well-formed* if *t* can be transformed into an empty trace in a sequence of steps, where in each step *i* a pair of events e_i^{rq}, e_i^{rsp} representing a (single) call of operation $op \in OL$ is removed from *t* (e_i^{rq} preceding e_i^{rsp} in *t*). For receiving a call of *op*, $e_i^{rq} = !op\uparrow$ and $e_i^{rsp} = ?op\downarrow$, for sending a call of *op*, $e_i^{rq} = ?op\uparrow$, $e_i^{rsp} = !op\downarrow$.

If such a sequence of the removal steps exists that in each step, e_i^{rq} immediately precedes e_i^{rsp}, *t* is a *non-overlapping* communication trace.

A PoSM is well-formed if all its communication traces are well-formed.

Theorem 7.14 If the syntactical definition of a PoSM P^A does not use explicit request and response ***PortTransitions*** (all ***PortTransitions*** are defined with the call transition and call state syntactical constructs) and each composite state in P^A may only exit with its completion event, then all traces from $LC(P^A)$ are well-formed. In addition, if P^A does not contain any orthogonal state, then all traces from $LC(P^A)$ are non-overlapping.

Proof sketch: A call transition (and also a call state) always specifies a pair of transitions labeled with the request and response events; unless the call transition (a call state) is contained in a composite state with a labeled outgoing transition (that could cause the call to terminate without the response event), the call always completes. □

If not contained in an orthogonal state, events for different calls may not interleave in trace *t*, until t_i is empty, there is always an e_i^{rq} immediately followed by a e_i^{rsp} in t_i. □

Claim 7.15 $LC(P^A)$ is a regular language.

Proof sketch: A PoSM P^A can be transformed to a finite automaton. By following the structure of P^A, orthogonal regions may be replaced with Cartesian product of states; a composite state can be replaced with its substates, redirecting outgoing transitions to all substates (except those that already have a higher-priority transition) and redirecting the incoming transition to the substate targeted by transition from initial ***PseudoState***. This way we yield a generalized non-deterministic finite automaton (employing empty transitions); the automaton generates a regular language. □

7.3. Composition Verification with PoSMs

Behavior Protocols [70] provide a *behavior compliance* relation, which can be used to verify composition of components based on their behavior specifications. In this section, we first briefly review behavior compliance as it is defined in Behavior Protocols [70] and describe how behavior compliance can be used to address consistency issues in the composition of software components. Afterwards, we show how behavior compliance can be applied to PoSMs. Finally, we discuss how this can be used to address the consistency issues in composition of UML 2.0 components.

7.3.1. Behavior Compliance in Behavior Protocols

> Although Behavior Protocols have been already described in detail in Sect. 4.1, we decided to preserve the flow of the PoSM paper [49] this chapter is based on; we include this brief description of behavior protocols exactly in the form as it was published in there. Note that the introduction in Sect. 4.1 used a slightly different notation, tailored for use with Pro-cases. Technically, both the definitions is the same; the notation used here is the notation used in the original paper on Behavior Protocols [70].

In behavior protocols, a single run of an agent *A* is captured as a sequence of atomic events (trace) from a finite domain *S* processed by *A*. Given a set of labels EventNames, *S* is formed as $\{?,!,\tau\} \times EventNames \times \{\uparrow,\downarrow\}$ (here, τ denotes an event internally processed by *A*; the symbols *?*, *!*, $\uparrow$ and $\downarrow$ stand for receive, send, request and response). Behavior of an agent *A* (denoted *L(A)*) is captured as the set of all traces of *A*, forming a language upon *S*.

Behavior of *A* may be described with a behavior protocol $Prot^A$, an expression syntactically generating a set of traces over S^* (denoted $L(Prot^A)$, conveniently a regular language). Employing a regular expression-like notation, behavior is described using event tokens for events from S and the following operators (given in priority order): ***** (repetition), **;** (sequencing), | (parallelism, based on arbitrary interleaving of traces), || (parallel-or, *A*||*B* is a shortcut for *A* + *B* + *A* | *B*) and **+** (alternative). Further, the composed operators are *composition* ($\sqcap_X$), *adjustment* ($|_X|$) and *consent* ($\triangledown_X$). The notation also uses the shortcuts discussed in Sect. 7.2.3 and parentheses.

Example 7.16 The behavior described in the PoSM notation in Figs. 7.1 and 7.2 may be expressed in the behavior protocols notation as:

```
?Clerk.deposit{ !DB.addBalance{ ?DBNotify.newBalance }
                |!Log.logDeposit }
```

The communication trace demonstrated in example 7.12, capturing this behavior, is also a trace of this behavior protocol.

Composition $Prot^A \sqcap_X Prot^B$ yields the behavior resulting when agents *A* and *B* described by protocols $Prot^A$ and $Prot^B$ are composed together; *X* is the set of event labels from $Events = EventNames \times \{\uparrow,\downarrow\}$ of events transmitted between *A* and *B*. For each pair of traces $\alpha \in L(Prot^A)$, $\beta \in L(Prot^B)$, the events from α and β arbitrarily interleave. In each such resulting trace, all events with label $x \in X$ are processed the following way: sequences of form *?x !x* or *!x ?x* are replaced by τx (an internal

event); a trace containing events with label *x* that cannot be processed this way (unmatched *!/?*) is discarded from the result of the composition operator.

The adjustment operator also interleaves pairs of traces $\alpha \in L(Prot^A)$, $\beta \in L(Prot^B)$, but exact match (not ?/! correspondence) of events with label from *X* is required and only pairs α,β that match on events from *X* are included in the resulting behavior.

The consent operator (introduced in [1, 2],) is similar to the composition operator, but generates *erroneous* traces for situations when interaction of A and B results into an error. The types of errors considered are *BadActivity* (A emits *a* but B is not ready to absorb *a*), *NoActivity* (similar to a deadlock situation) and *Divergence* (interaction of A and B never stops). The consent operator implicitly provides a relation for checking the composition of A and B, by considering the composition to be correct if $A \triangledown_X B$ contains no erroneous traces.

Definition 7.17 We assume the set *S* is divided into disjoint sets S_{prov} (inputs, events on provided interfaces) and S_{req} (outputs, events on required interfaces). Behavior *L(A)* of agent *A* is *compliant* with behavior $L(Prot^A)$ of protocol $Prot^A$ on set *S* if (i) *A* can accept any sequence of inputs dictated by $Prot^A$ and (ii) for such inputs, *A* creates only outputs anticipated by $Prot^A$.
A formal definition is provided via the adjustment operator: *L(A)* is compliant with $L(Prot^A)$ on set *S* iff:

(i) $L(Prot^A) / S_{prov} \subseteq L(A) / S_{prov}$

and

(ii) $(L(A) / S \,|_{Sprov}|\, L(Prot^A) / S_{prov}) \subseteq L(Prot^A) / S$

where / is the operator for restriction.

By adjusting *L(A)/S* with $L(Prot^A) / S_{prov}$ (the dictated inputs) over S_{prov}, only traces from *L(A)/S* with inputs dictated by $L(Prot^A)$ are considered; these traces must be contained in $L(Prot^A / S)$. For reference, the original definition of behavior compliance is available in [70]. The case study in Sect. 7.4 provides demonstrations of the behavioral compliance relation.

7.3.2. Composition Verification with Behavior Compliance

In Sect. 2.3 (and in [67, 69]), we identified the consistency issues to be considered in component composition in a hierarchical component model. Basically, the issues are: (a) whether the *composed behavior* of components $A_1..A_n$ forming together component S is compliant with the behavior specification for S; (b) whether two distinct specifications for a component specify “compatible” behavior; (c) and whether communication between A and B is correct.

In behavior protocols, the issue (a) is addressed by the compliance relation (employing the composition operator to obtain the composed behavior). The compliance relation may be also used to address the issue (b). Finally, the issue (c) is addressed by the consent operator.

Note that a verifier tool [46, 77] is available to test the compliance relation (supporting the composition operator); thus, the issues (a) and (b) are decidable in behavior protocols. Enhancing the verifier tool to support the consent operator is subject of future research.

7.3.3. Behavior Compliance in PoSMs

The behavior protocols compliance relation is defined on languages (upon the domain of communication events) and thus, its definition is applicable to PoSMs as well. The set *CE* (domain of communication events) can be used in place of the set *S*. The set S_{prov} is the set of events on provided interfaces of the ***Port*** the PoSM is associated with, S_{req} is the set of events on required interfaces.

Although composition and consent are protocol operators, their semantics is defined solely based on the languages generated by their operands and thus, their definition can be extended to communication languages of PoSMs.

Therefore, the consistency issues (a), (b) and (c) can be addressed for PoSMs; the existing behavior protocols compliance verifier may be employed to evaluate the compliance relation on PoSMs.

Note that the compliance relation is applied only to communication languages generated by PoSMs; neither the states, nor the structure of the state machine are considered in the compliance relation. Broadening the definition of compliance relation to execution languages is subject of future research.

7.3.4. Relation of Behavior Protocols and Port State Machines

Both PoSMs and behavior protocols describe behavior in a way that yields a set of communication traces, conveniently a regular language. It is possible to transform a behavior protocol into a PoSM, i.e., construct a PoSM generating the same communication language as the behavior protocol (restricted to the communication events contained in *S*).

In this process, (i) an event token explicitly specifying a request (↑) or a response (↓) is translated into an explicit ***PortTransition***, (ii) a shortcut *?a* or *!a* is translated into a call transition, (iii) shortcut *?a{Prot}* or *!a{Prot}* is translated into a call state; the protocol Port is transformed into the internal behavior of the call state. A protocol may also specify internal events (τ), which do not influence the communication described by the protocol and are neither considered in the compliance relation; we omit them in the transformation. Following the syntactic structure of the protocol, we translate the sequencing operator (**;**) into sequenced states, repetition (*****) into a loop transition, alternative (**+**) into multiple outgoing transitions; parallelism (|) is modeled via orthogonal regions.

Note that in this process, we create states as necessary to transform the structure of the protocol into a state machine. In the PoSM shown in Figs. 7.1 and 7.2, for selected states, names are provided to make the PoSM specification more expressive. In an automated process, anonymous states (without a name) have to be used instead. Here, automatically generated state labels may be employed to distinguish states in execution traces (in a way similar to how the states `ab1` or `logfin` are labeled).

In a similar vein, we may consider constructing a behavior protocol for a Port State Machine. However, in the general case, the only solution is to first transform the state machine into a regular automaton (expanding composite states) and afterwards, apply the generic algorithm for transforming a regular automaton into a regular expression. Such a process would significantly impair readability of the resulting behavior protocol.

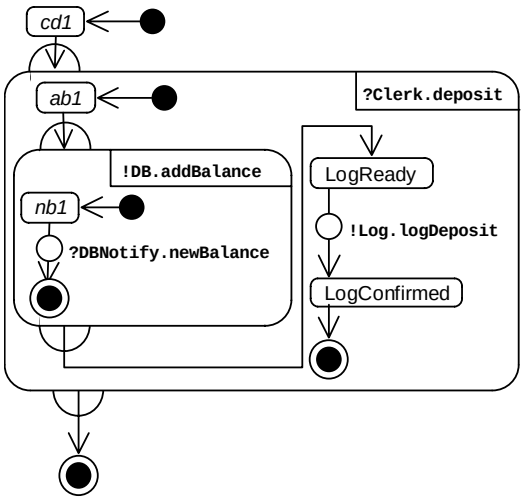
There may be interesting special cases, namely, when the only composite states used in the state machine are call states. Here, the transformation can be done

separately at each level of nesting; exploring these special cases is subject of future research.

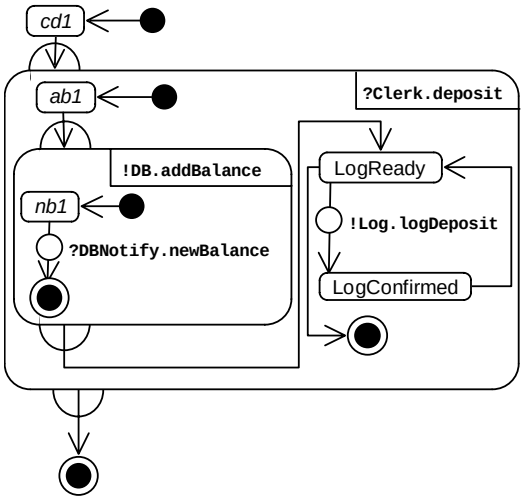
7.4. Case Study: Compliance of Port State Machine

The definition of the behavior compliance relation is based on the notion of *substitutability* [70]; in the SOFA Component model [66], behavior compliance is used to verify composition of software components. Given the specification of behavior of a component in the form of a *frame protocol*, the key question is, whether behavior of the realization of the component, as described by its *architecture protocol*, is compliant with the *frame protocol*.

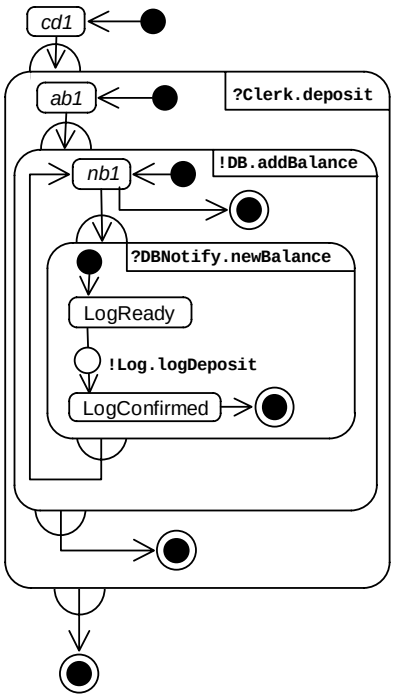
This may be also applied to verify composition of UML 2.0 components; with PoSMs, we may reason on compliance of a realization described by a PoSM P_i^R with the specification described by PoSM P^S.



(a) !Log.logDeposit in sequence



(b) !Log.logDeposit called zero or more times



(c) !Log.logDeposit called inside ?DBNotify.newBalance (which may be called zero or more times)

Figure 7.4: Port State Machines describing possible realizations of the behavior specified by Fig. 7.2

Let us consider the behavior specified by PoSM in Fig. 7.2 as the specification PoSM P^S. This PoSM specifies that while a call **`?Clerk.deposit`** is being processed, a call **`!Log.logDeposit`** is issued in parallel with issuing a call to **`!DB.addBalance`**, during which a call **`?DBNotify.newBalance`** is received. Note that here, "in parallel" means arbitrary interleaving of the traces generated by the orthogonal regions of the PoSM.

Example 7.18 Figure 7.4 shows PoSM specifications of three possible realizations of this specification. In Fig. 7.4 (a), PoSM P^{R_1} specifies that the call **`!Log.logDeposit`** occurs after the call **`!DB.addBalance`** completes. Such realization is (trivially) compliant with the specification, its behavior restricted to S_{prov} contains all traces from $LC(P^S) / S_{prov}$ (condition (i) of Def. 7.17) and $LC(P^{R_1}) \subseteq LC(P^S)$, thus condition (ii) holds as well.

Example 7.19 The PoSM P^{R_2} in Fig. 7.4 (b) in addition specifies that the call **`!Log.logDeposit`** may occur zero or more times (instead of exactly once). Consequently, its language $LC(P^{R_2})$ is not compliant with $LC(P^S)$ – although condition (i) of Def. 7.17 holds, condition (ii) does not: $LC(P^{R_2})$ contains traces capturing an arbitrary number of **`!Log.logDeposit`** calls, while $LC(P^S)$ contains only traces where the call **`!Log.logDeposit`** occurs exactly once.

Example 7.20 The PoSM P^{R_3} in Fig. 7.4 (c) instead specifies that the call **`?DBNotify.newBalance`** may be processed zero or more times; and that the call **`!Log.logDeposit`** will be issued while processing **`?DBNotify.newBalance`**, each time this call is received. Surprisingly, $LC(P^{R_3})$ is compliant with the $LC(P^S)$. Although the P^{R_3} can call **`!Log.logDeposit`** more than once (or not at all), this may occur only in runs where **`?DBNotify.newBalance`** is called more than once (or not at all). Thus, after reducing (via the adjustment operator) $LC(P^{R_3})$ to traces with inputs contained in $LC(P^S)$ (i.e., traces where **`?DBNotify.newBalance`** is called exactly once), the resulting behavior calls **`!Log.logDeposit`** exactly once (in an order permitted by the P^S) and therefore, condition (ii) of Def. 7.17 holds; condition (i) holds trivially.

As the last example demonstrates, behavior compliance permits that behavior of a realization contains traces with outputs not expected by the specification, in case such traces result from inputs not permitted by the specification. Consequently, substitutability based on behavior compliance permits a broader set of realizations to be used for a given component specification.

7.5. PoSM as a Use Case Specification Mechanism

Port State Machines, being able to capture the communication of an entity (component) with entities (components) it is connected with, can be also employed as a notation for use cases. In this section, we employ the Trace-based UC View developed in Chapter 6 to analyze how PoSM support behavior assembly, behavior composition and how they address the issues in consistency reasoning. We use subscript PoSM to denote instances of Trace-based UC View specific to Port State Machines.

Scenarios: A scenario of a PoSM is a trace, as defined in Sect. 7.2.2. As the compliance relation is defined upon communication languages of PoSMs, we choose communication

language as the interpretation of Scenarios. Thus, Act_{PoSM} = *CE* (the domain of communication events and $Scenarios_{PoSM} = Act_{PoSM}$*, the behavior of a use case specified as a PoSM is defined $Com_{PoSM}(UC_{PoSM}{}^{A}{}_{j}) = LC(P^{A}{}_{j})$, where $P^{A}{}_{j}$ is the PoSM behavior specification of $UC_{PoSM}{}^{A}{}_{j}$.

Assembled behavior: In order to define semantics of use case expressions for PoSM, each operator from $\underline{OP}$ has to be associated with an operation θop; as scenarios are traces, such an operation can be found as operation upon sets of traces for all operators from $\underline{OP}$ (i.e., +, ; * and |) and thus, $OP_{PoSM} = \underline{OP}$.

Representative use case: Same as for StateMachines, a PoSM can include another PoSM via a submachine state. We employ this to define functions Alt, Seq, Rep and Par; these functions implement the operators from OP_{PoSM}. The functions are defined in a way similar to the functions defined for StateMachines, each constructs a new PoSM containing submachine states referring to operands of the function. The submachine states are lined from an empty simple state (Alt), joined via an unlabeled ProtocolTransition (Seq) and with a loop unlabeled ProtocolTransition (Rep). The function Par, implementing the parallel composition operator "|", creates an orthogonal composite state; each of the orthogonal regions contains a submachine state referring to an operand of the function. Due to the atomicity of the events (request/response), a representative use case can be created for all operators from $OP_{PoSM} = \underline{OP}$, including parallel composition.

Composed behavior: We interpret $Msg_{PoSM}(A)$ as the set of communication events *e* (*e* ∈ *CE*) that are used as the label of a transition in a PoSM P^{A}. We define $Pair_{PoSM}{}^{A,B}(e^{A},e^{B})$ iff $e^{A} = < send, label_{op}, ocp_{T} > \wedge e^{B} = < receive, label_{op}, ocp_{T} >$ or vice versa, i.e., $e^{A} = < receive, label_{op}, ocp_{T} > \wedge e^{B} = < send, label_{op}, ocp_{T} >$. As scenarios are traces and the $Pair_{PoSM}{}^{A,B}(e^{A},e^{B})$ relation is defined, the composed behavior concept is interpreted in PoSM.

Consistency reasoning: The behavior compliance relation defined for behavior protocols applies also to the communication language of Port State Machines; further, the consent relation applies as well. As assembled behavior can be obtained for all operations from $\underline{OP}$ and composed behavior can be obtained, Port State Machines fully address the consistency issues articulated in Sect. 6.4.4.

7.6. Evaluation

Port State Machines permit to capture the interleaving of events (representing operation calls) on a set of provided and required interfaces associated with a ***Port*** of a UML 2.0 ***Component***. PoSMs support modeling nested calls; technically, an arbitrary fixed depth of recursion can be modeled with a PoSM. Unlimited recursion (which inherently causes the generated language not to be regular) is avoided.

Conveniently, the language generated by a PoSM is regular (taking into account that there are no constraints, no event deferring and, inherently to state machines, no recursion). Thus, PoSMs permit to establish a compliance relation and apply the behavior protocols compliance verifier [46, 77].

The UML 2.0 ***Interactions*** (former sequence diagrams) also explicitly capture an operation call with atomic request and response events; also, trace model semantics is defined for ***Interactions***. However, Interactions focus on describing communication among interconnected objects. Although it is possible to employ a ***formalGate*** to capture calls of an ***Operation*** exposed via an ***Interface*** of a ***Port***, the notation does not support efficiently describing the ordering of communication on a Port.

7.7. Conclusion (Chapter Summary)

In this paper (chapter), we proposed the Port State Machines (PoSMs). Building on UML 2.0 [10] Protocol State Machines and Behavior Protocols [70], Port State Machines allow to capture the interleaving of operation calls on a set of provided and required interfaces. Operation calls are captured as a pair of atomic events representing the start of the call (request) and end of the call (response). This way, nesting of operation calls (e.g., a call-back) can be captured in a specification.

Moreover, as PoSMs use atomic events, the behavior on a Port specified by a PoSM is captured as a set of traces, forming a language upon a finite alphabet. The behavior compliance relation has been established to reason on compatibility of PoSM specifications. As the language of a PoSM is regular, the compliance relation is decidable; conveniently, an already existing verification tool [46, 77] can be employed for this task. Further, as evaluated in Sect. 7.5, PoSM, when considered as an instance of Trace-based UC View, address the consistency issues and can be used to model use cases.

Chapter 8

Related work

In this Chapter, we analyze the related work in the fields relevant to the research described in this thesis. As there is a significant overlap of topics between the papers this thesis is based on, and as each of the papers describes related work in several related fields, we decided to organize the related work analysis into several sections based on the field; each section contains excerpts from multiple papers. As elsewhere in this thesis, excerpted text is marked by using a sans-serif font; we denote the origin of each such excerpted description with a margin note.

8.1. Formal Methods in Behavior Specifications

gettingWP+ posm

In [82], Message Sequence Charts (MSC) are translated into an LTS which can by analyzed by model checking for deadlock, safety, and liveness properties. The concept of High-level MSCs (hMSCs) composed of basic MSCs (bMSCs) resembles our concept of use case expressions and a synthesis and analysis algorithm is provided. However, only a single level of composition is allowed; moreover, the set of operations is weaker than in our approach (e.g., parallel composition, compliance and consent are not considered). Further, as the approach is focused on individual messages rather than on operation calls, call nesting is not addressed here.

posm

The work presented in [42] defines an equivalence relation for state-chart specifications; bisimulation of labeled transition systems is used for testing the equivalence. In [41], the authors translate UML statecharts into PROMELA, the input language of SPIN. In a way similar to our approach, a subset of statecharts is chosen such that the statechart can be translated to a finite state automaton. However, call nesting is not considered in this approach.

posm

In [81], two algorithms for testing conformance of LTS and behavior expressions are presented. The approach employs test cases, testing is done via synchronous parallel execution of a test case and the implementation. The test cases considered are deterministic, but the implementation may behave nondeterministically; thus, as an implementation passes a test case only if all possible runs pass, theoretically, a test case may have to be executed infinitely many times.

Live Sequence Charts (LSC) [16] extend Message Sequence Charts (MSC) with the concept of liveness, identifying mandatory parts of behavior (“things that must occur”). In principle, this is achieved by distinguishing between existential and universal sequence charts; similar conceptual distinction is also done at finer granularity of a specification: messages, locations and conditions. The motivation for this distinction is that at an early stage of requirement specifications, the sequence charts (playing the role of use cases) specify samples of possible behavior (“what may happen”), while at a later stage, after refining the requirements specification, the semantics of sequence charts specify mandatory behavior. In LSCs, mandatory behavior applies to all scenarios if the sequence chart is triggered (its *pre-chart* matches a sequence of events in the scenario and a boolean *activation condition* holds). Employing the trace-based semantics of LSCs, authors consider

several approaches to verification. Monitoring of simulations based on pre-existing input creates a watchdog for each triggered sequence chart; here note that a sequence chart may be triggered more than once at a single point in the simulation. In addition, authors also consider deriving input stimuli for the simulation from the LSC specification. Further, authors also consider model-checking; this his however postponed to a follow-up paper.

The approach proposed in [76] combines pre- and post-condition based operation schemas with behavior specifications ordering actions (expressed via use cases or UML protocol state machines). Authors recognize that pre- and post-conditions are not widely used in practice. Proposing modifications to OCL, authors suggest employing a formalism not too heavy to learn and use, and complementing use case descriptions with pre- and post-condition descriptions. This permits reasoning about system properties and provides a basis for testing and debugging, as well as for getting a predictable decomposition level on which one can base a systematic transition to design.

In [20], authors propose a consistency workbench for custom verifications of UML models. Developers may store consistency problems in a consistency problem catalogue, define transformations of UML models to CSP as well as consistency checks to be performed. A model checker can be then employed to execute the consistency check for a transformed model. A set of graph transformation rules has been developed to translate statecharts to CSP; a consistency check has been specified to validate consistency of a protocol statechart with a statechart describing an object implementation.

8.2. Behavior Specifications in Software Engineering

uml20uc+posm

In [80], the authors analyze how behavior can be specified via composition of UML ***StateMachines***. Due to the run-to-completion semantics and due to reactions encapsulated in ***Activities*** associated with a transition, ***StateMachines*** cannot model recursive calls; an attempt to do so will lead to a deadlock. The problem is addressed by introducing *method state machines* (MSMs). They implement method bodies as state machines containing special constructs, *invocation boxes*, referring to methods implemented by other MSMs. This extension of state machines also allows to interpret assembled behavior (all operations from *OP*). The Method State Machines also extend state machines with the ability to model recursion. Recognizing the obstacles of the run-to-completion semantics, the authors model operation calls with two events, corresponding to request and response. A relation of compliance of a Protocol State Machine with a set of MSMs is defined; however, as a tradeoff for modeling recursion, the relation is not decidable. Moreover, the approach taken there is object-based, focused on the graph of operation calls among cooperating objects; it would not be possible to capture external communication on the interfaces of a software component with MSMs without a significant modification.

uml20+posm

In [86], the author aims at improving the support provided by CASE tools in the software development process. He points out that UML behavior specification mechanisms have “unclear semantics” and specify “partly redundant and partly complementary information (without underling formal approach)”. He also identifies lack of formal checks for consistency among different behavior specification mechanisms. Options are analyzed for automating the creation of class diagrams

from a UML use case model; an approach for generating StateMachines from sequence diagrams is described (implemented in the Fujaba tool [86]); here, the work also aims to check for consistency. However, neither composition, nor assembly is addressed here.

gettingWP

The authors of [85] performed a study of readability of formal requirements, evaluating how the presence of certain features in a notation influences readability of the notation. It is emphasized that readability is crucial for acceptance of formal methods by the industrial community; we believe that our Pro-cases meet this requirement well.

The design methodologies recommended in literature [37, 40, 18, 26] consider the software development process from various aspects. Larman [40] focuses on developing the design model of the system. The recommended way to obtain the interfaces of the system under design is to transform use cases into system sequence diagrams; due to the limitations of the sequence diagrams visual notation, the behavior specified by use cases can be translated into sequence diagrams only in a limited way. Graham [26] provides a exhaustive analysis of approaches available; there, use cases are considered as only one option. It is highlighted that use cases cannot capture all of the requirements and must be accompanied by additional requirements documents.

The Catalysis approach [18, 17] of D'Souza focuses on reliability of the software development process; traceability of requirements is considered to be a key issue. Textual use cases are considered, however, for precision, pre-conditions and post-conditions are recommended for refined specifications.

The KISS method [37] proposed by Kristen is based on creating an object oriented design in a conceptual model independent of an implementation language, and thus not bound by restrictions of a specific implementation environment. The method puts focus on precise specifications, which also permit validating quality of specifications and models. With this method, large parts of the final information system being built can be generated from the model via transformation rules.

8.3. Use Cases and Relations among them.

gettingWP

In [78], it is emphasized that the use case concept in UML has not reached the point where a tool could support the use of use cases without making major decisions concerning how to interpret the standard. It basically argues that use cases should be formalizable in a soundly-based, tool-supportable way, in order to relate use cases to the design of a respective system. One of the ways of formalizing use cases in UML is to formally interpret them as sequences of actions. Here, to avoid possible limitations imposed by choosing a concrete process algebra, a labeled transition system (LTS) is recommended; a high-level formalism is intentionally avoided. It is highlighted that UML state machine should not be used to realize an LTS, due to its extra power with respect to LTS and the correspondingly complex semantics. Compared to this, our approach in Generic UC View is to capture the generic properties of use cases (and the behavior they describe) in such a way that a variety of concrete instances (Concrete UC Views) can be defined later on, including an LTS (which our Pro-cases in principle are), each of them featuring these generic properties.

posm

An abstract state machine language is employed for writing use cases in [27]; instead on reasoning on behavior compliance, the authors aim to generate test scenarios from the abstract state machine specification; selecting test sequences is also considered in [30].

uml20uc
(work in progress)

The UML (1.4) view of *use cases* is also analyzed in [22], however, the authors rather focus on use case relations (include, extend), claiming that these are not defined precisely enough to avoid ambiguities and misunderstandings.

In his book [13] focused on textual use cases, Cockburn elaborates the concept of SuD, employing *in/out lists* to identify the entities contained in the SuD; the process eventually also yields the list of potential actors. He also identifies that an entity being an actor in a set of use cases can be the SuD in another use cases; in such a situation, the entity that was SuD in the original use cases may become an actor in the other model. Nesting of entities is also considered; Cockburn suggests to write an *outermost* use case to provide an entry point to its subordinate use cases. Further, *goal-levels* are used to differentiate between use cases; the include relation should link only use cases from adjacent but different levels (specific exceptions are permitted and justified). A *summary* use case, written at the highest goal-level, provides an alternative approach to obtaining the whole picture of an entity's behavior. While Cockburn touches the issues of hierarchy of entities and obtaining the whole picture of behavior, compatibility of behavior among different levels of nesting is not considered.

Further, Cockburn clearly defines structure of use cases (main success scenario, variations and extensions); however, some issues of semantics are not specified (sect. 5.4), assuming they will be obvious from the context in a particular use case model. Further, Cockburn provides guidelines for writing the steps and conditions of textual use cases; we employ these guidelines in the transformation proposed in Chapter 5.

gettingWP

An interesting approach is taken in [29] where a use case is represented as an activity diagram and for each action in it, a graph transformation rule is defined upon collaborations of the objects involved in the action. Conflicts/dependencies between actions in different use cases (UC^{A}_{j} and UC^{A}_{k} in our terminology) can be identified.

uml20+posm

Use Case Maps (UCM) [4, 9] employ a visual notation to capture the trajectory of a use case through the hierarchy of a system. Similar to *use cases*, UCM models behavior by capturing significant/typical scenarios. Instead of assembling behavior, UCM puts focus on describing behavior at multiple levels of entity nesting and visualizing such scenarios in a single diagram.

With respect to the structure captured in the diagram, UCMs *"project the scenarios of use case onto solutions"*. Responsibility points, touching paths, show responsibilities of components, their order gives a causal sequence. The notation for paths supports forks and joins (with the semantics of logical AND and OR respectively); notation extensions considered include timed waiting places, which can be used to represent optional timeouts. Further, the way UCM captures scenarios as a typical walkthrough can be enriched with failure points and failure handlers.

Note that for a component, use case maps show the nesting of calls in a scenario. Nonetheless, as use case maps are focused on individual scenarios, obtaining the "whole picture" of behavior on the interfaces of a component is not possible.

uml20uc
(work in progress)

The analysis of support for use cases in UML 1.x in [32] also identifies the need to distinguish between "complete" and "fragment" use cases, reflecting the way we select the set of relevant use cases in the definition of a use case model. Further, this work focuses on relations between a use case and its scenarios, concluding that the use case should be associated with a set of scenarios. Interestingly, this work also identifies the need to capture the connections among entities involved in a use case (SuD and actors); the proposed *use case units* are similar to the scope diagrams (Sect. 2.2).

The paper also articulates the following issues identified in UML 1.x (1.5 in particular). First, in the *Use-Case Class/Instance* issue, authors highlight that in UML, use case is a class (Classifier) and scenarios are its instances. The author's opinion is that instead, a use case should be an object (an instance), associated with the scenarios it generates. Further, the paper identifies the *Actors Call Use case (ACU) conjecture*: The terminology used in UML frequently states that *"an actor calls a use case"*. While the actor actually calls operations on the system specified by the use case, the use case (being actually only a behavior specification) is used in the role of the entity executing the behavior. Finally, the *Use-Case Execution Control* issue discovers that the use case execution semantics does not specify who chooses which use case and which scenario to select (i.e., it does not specify whether it is an internal or external choice with respect to *SuD*); therefore, simulation of use cases is not possible. The issues identified in this paper are related to the issues we identified in our analysis of UML 2.0 in Chapter 6 and in particular in Sect. 6.6 focused on the UseCases package.

8.4. Processing Specifications in Natural Language

natlang
(work in progress)
(whole section)

To ease the task of processing natural language documents, a well-established approach is to employ a *controlled language* for writing specifications, restricting the grammar of a natural language (NL) to use only certain word forms and sentence structures, as well as reducing the vocabulary. A well-known industry standard, used in the aviation industry, is the AECMA Simplified English; the term Simplified English is also often used for controlled languages in general. There, the goal of controlled languages is to ease understanding of a document for non-native English readers, as well as facilitate unambiguous translations into other languages.

Understanding natural language use cases is explored in [73], with the goal to discover relations among concepts. The authors also employ a natural language parser, to ease the work of the parser, they propose a controlled language with simple rules on word forms and sentence structure, which coin the same principles as the general guidelines for writing use cases proposed elsewhere; a tool is implemented to aid the writer in conforming to the restrictions imposed by the controlled language.

A Two-Level Grammar (TLG) is used in [44] to translate natural language requirements into a knowledge base (in TLG), which is subsequently translated into VDM. This is further used in [83] for automatic extraction of QoS requirements from natural language specifications; the approach employs the complete domain vocabulary and a tailored parsing scheme.

The CICO domain specific parser [25] is used in [23] and [24] to discover dependencies among actions and relations and between actions and values in the

system. Contrary to our approach, CICO uses domain specific rules and annotations.

The Simple Interfacing (SI) framework described in [36] puts focus on creating user interface from use cases, with the intention to verify that implementation is consistent with requirement specification; the approach assumes direct translation of natural language use cases into SI code.

The authors of [21] employ linguistic techniques to analyze quality of use cases. Criteria and metrics for evaluating the quality of use case specifications are defined; defects measured include, e.g., *vagueness* and *unexplanation*. The goal is to evaluate quality of use cases and check their conformance with the Simplified English controlled language.

The work in [38] targets checking consistency of natural language requirement specifications with UML designs. Instead of employing a natural language parser, the parsing code is captured as axioms in a knowledge base. An interesting point in this approach is that the lookup table (for matching natural language terms and UML model classes) is constructed based on already discovered matches and the UML design specification (finding the most appropriate match).

In [65], a controlled language and a rule-based parser are used to analyze NL requirements with the goal to assign Logical Form (LF) to NL requirement specifications; focus is put on resolving parsing errors and ambiguities. In the controlled language selection, it is pointed out that a too restrictive controlled language may be irritating to use (and read), as well as hard to learn to follow.

Chapter 9

Conclusion, Evaluation & Future Prospects

In this concluding chapter, we summarize the achievements of this thesis. In Sect. 9.1 we evaluate the achievements of thesis, with particular focus on how the goals set in Sect. 2.4 were met; we discuss the prototype implementation and our practical experience in Sect. 9.2. We lay out future work in Sect. 9.3 and finally articulate a conclusion in Sect. 9.4.

9.1. Evaluation

In this section, we evaluate the achievements of this thesis, in particular with respect to the goals set for the thesis in Sect. 2.4. In Sect. 9.1.1, we evaluate the Generic UC View, its application to analysis of existing use case approaches, and the proposed Pro-case notation, addressing the goals (i), (ii), (iii) and (iv). We summarize our results on addressing goal (v) to convert textual use cases into Pro-cases in Sect. 9.1.2. In Sect. 9.1.3, we sum up our analysis of UML 2.0 behavior specification mechanisms, the goal (vi) of the thesis. In Sect. 9.1.4, we review how the concluding goal (vii) was addressed by proposing the Port State Machines.

9.1.1. Generic View on Use Cases

gettingWP

In this section, we review how the goals (i), (ii), (iii) and (iv) of the thesis were met by Chapters 2, 3 and 4; in the review of the Generic UC View, we also evaluate how it meets the goals (1), (2) and (3) outlined for it in Sect. 2.1.

Meeting the goal (1) articulated in Sect. 2.1, we presented Generic UC View, targeting use case modeling. The message of this model is as follows: (i) For very basic modeling define a domain Scenarios. If an includes relation on use cases is desirable then define also a subscenario relation upon Scenarios as it helps define semantics on includes clearly. (ii) It has to be stated without any ambiguity how the scenarios of a use case UC^A contribute to scenarios in Com(A). Therefore the Generic UC View distinguishes the set U^A of all use cases of A and a use case model UM^A of A as the set of use cases where “the scenario generations start”. (iii) The whole picture behavior of A (Com(A)) can be infinite, it is desirable to “see” it as a single use case – representative. This can be advantageously done when use case expressions are defined. (iv) To reason on specification consistency of cooperating entities, in both nested an “sibling” position, Generic UC View articulates three relations/operations to be defined.

Addressing the goal (2) we evaluated the UML support for use cases in terms of Generic UC View. Here the bottom line is that UML does not clearly address (ii) above; this in principle means that it is not clear what the whole picture behavior of an entity A really should be if there are more use cases written for A. Being very generic, UML does not consider reasoning on use case specifications.

As a proof of the concept, protocol use cases (Pro-cases), based on behavior protocols [70], have been introduced (addressing goal (3)), which support all the features (i) - (iv) above. Moreover, as they are based on in principal regular languages, all the relations and operations introduced in Generic UC View are decidable. To summarize the benefits from creating a Pro-case model:

(I) With use case expressions, behavior can be assembled to form a single Pro-case as the "whole picture" of an entity's behavior, in a readable and comprehendible notation;
(II) Parallelism can be captured, creating a more precise specification;
(III) Using the decidable compliant with relation, the consistency issues (a), (b), (c) can be addressed;
(IV) Pro-cases can be helpful in an early stage of design – showing the interactions of an entity, they can be useful when assigning responsibilities to classes, identifying operations, behavior (structure) of methods; thus, Pro-cases can serve as a powerful replacement of system sequence diagrams [40].

9.1.2. Conversion of Use Cases to Behavior Specifications

While Pro-cases provide the advantages discussed above, there is still justification to use textual use cases. To avoid duplication of effort in developing Pro-cases and textual use cases in parallel, we have proposed a scheme for converting textual use cases into Pro-cases, following simple guidelines demonstrated in Sect. 4.3. Further, in Chapter 5, we have employed readily available linguistic tools to automate this conversion. The proposed approach is implemented in the *Procasor* tool; we have evaluated feasibility of our approach by applying the tool to a set of use cases developed within our previous work on the formal model Generic UC View (published in [68] in Nov 2002).

9.1.3. UML 2.0 Analysis

With the intension to analyze the emerging standard UML 2.0, we have extended our generic use case model to utilize the features of a behavior specification mechanism where semantics is defined based on traces, while preserving support for other behavior specification mechanisms as well. As extensions to *Generic UC View*, we have proposed the specialized models *Basic UC View* and *Trace-based UC View*.

In these extensions, we refine the assembly operations already considered in Generic UC View as the operations for use case expressions. In our analysis of UML 2.0, we have explored the options to interpret the assembly operations via native operations of its behavior specification mechanisms, employing their well defined structure and syntax.

The highlights of our findings are that Interactions define trace-based scenarios and support behavior assembly and composition; although semantics of ProtocolStateMachines can also be interpreted with traces, behavior composition is not possible. Behavior of Activities and State Machines cannot be interpreted with traces and their support for behavior assembly is limited. In conclusion, only Interactions satisfy the prerequisites for reasoning on behavior specified in multiple use cases of an entity.

9.1.4. Port State Machines

Having identified that UML 2.0 StateMachines cannot capture interleaving and nesting of operation calls, we proposed Port State Machines to provide a remedy to this problem. By representing an operation call with two separate events, *request* and *response*, Port State Machines are able to capture the interleaving and nesting of operation calls of provided and required interfaces of a Port, encapsulating the communication of a UML 2.0 component. Conveniently, due to the restrictions imposed on Port State Machines, the language representing the behavior specified by a Port State Machine is a regular language. Consequently, the behavior compliance relation defined for languages of behavior protocols [70] may be used for languages of PoSM as well; the behavior compliance relation is decidable and a verifier tool is available [46, 77].

9.2. Implementation and Practical Experience

gettingWP

To proof the concept of deriving (manually) behavior specifications from use cases; we have performed a case study on a use case specification of a sample marketplace application. We have also employed these methods in a graduate-level object oriented methodology course; students were instructed to use Pro-cases instead of System Sequence Diagrams in the transition from use cases to the initial design.

(1) As a proof of the concepts, we (re)designed a part of a project featuring nested entities (Fig. 4.1); Marketplace Information System (*M*) yielded 7 use cases, while the internal three entities 2, 8 and 2 use cases. About 20 min were required to transform a textual use case (written using a unified template [13]) into a Pro-case. To assemble the behavior into frame Pro-cases, the time requirement was about 10 min for each of the four components. An important side-effect of this process had been realizing that one of the Pro-cases of *M* can be added to its assembled behavior via parallel composition, which in principle means that we have added 8th (very complex, describing all potential trace interleavings) use case to the original behavior description of *M*. Compliance of the composition of the three internal frame Pro-cases with the frame Pro-case of *M* was done by the SOFA Protocol Verifier [46] (written in Java), finding three inconsistencies in the original specification.

(2) In a major group assignment, students taking their first software design course were instructed to transform their use cases into Pro-cases. It took them only about 15 minutes to get the behavior protocol idea. Our observation had been that students had no difficulty creating Pro-cases and actually preferred them very much over System Sequence Diagrams [40]. Furthermore, as the requirements were specified more precisely, the projects progressed more smoothly compared to a previous run of the course where classical data-driven design was used.

We have implemented the conversion of textual use cases into Pro-cases in the *Procasor* tool. The tool employs existing linguistic tools, in particular the statistical natural language parser developed by Michael Collins [15], the Maximum Entropy tagger developed by Adwait Ratnaparkhi [72] and the morphological tool morpha developed by John A. Carroll et al. [56]. For each sentence describing a use case step, the tools yield a parse tree annotated with POS-tags and the lemmas of the words (as shown in Fig. 5.2. The java-based Procasor tool subsequently processes the parse trees, acquires the principal information describing the action, constructs an event-token for each step, and eventually translates the use case into a Pro-case.

We have applied the Procasor tool to the Marketplace use case model [68] developed within our theoretical work on the Generic UC View formal model. The use case model was developed without considerations for automatic processing, only following the generic guidelines recommended by Cockburn [13]. Only minor modifications (mostly typo corrections) were necessary for the tool to process the textual use cases; the Pro-cases obtained were very close to Pro-cases originally obtained by hand in [68]. The prototype implementation, as well as the experiment with the Marketplace use case model, are described in detail in Sect. 5.5; for the experiment, the input, intermediate and output data are in Appendices B, C and D respectively.

9.3. Future Work

In this section, we analyze prospects for future work building upon the results already achieved and described in this thesis. We split the section into three parts, focusing on future work (i) in the use cases and Pro-cases, (ii) employing linguistic tools to automate the conversion of textual use cases to Pro-cases, and (iii) Port State Machines.

9.3.1. Future Work on Use Cases and Pro-cases

gettingWP

In use cases, failure handling is an important issue. In the current state, Pro-cases allow to express extensions as alternatives, with the condition captured as an internal event. In the case of failures inside nested blocks, the readability of the Pro-case can be negatively influenced. Our future work aims at enhancing behavior protocols with exceptions, to permit a more elegant way to handle error conditions. Our intention is to preserve the current readability and lightweight nature of the notation; the key point is to avoid turning the notation into process equations. Moreover, erroneous traces are being investigated [1] to capture inconsistencies in composition.

To provide a proof-of-the-concept for behavior assembly, we plan to design UML extensions supporting use case expressions, possibly in the form of a UML Profile; the extensions (profile) will be implemented for a UML tool. With use case expressions, the extensions will permit to capture the assembled behavior of a use case model. The extensions will be integrated with the Procasor tool to convert textual use cases into Pro-cases (more on this in Sect. 9.3.2). The existing behavior compliance verifier tool [46, 77] will be integrated into the extensions as well to reason on consistency of models specified with Pro-cases. Moreover, PoSMs will be included in this extension, also employing the verifier tool to reason on consistency of PoSM specifications.

9.3.2. Future Work in Employing Linguistic Tools

natlang
(work in progress)

There is a strong potential in developing interactive tools employing the conversion described in this paper. In an interactive use case writing tool, the event token might be offered immediately after a use case step is written; here, feedback from the user on the quality of the parse might be used in subsequent processing. Also, the tool might offer several parses (highlighting the most probable ones). Here,

we consider employing the new David Bikel's parser [7] featuring *n-best* parsing. Based on the parse(s) offered, the user might instead rephrase the sentence, improving the clarity of the use case. Also, the interaction of the user and the tool might facilitate maintaining event tokens consistent across the use case model.

While we originally focused on transforming a textual use case into a Pro-case (a behavior protocol, thus a regular-like expression in principle), transformation to other message-oriented behavior specification mechanisms may be feasible as well. As the transformation yields a finite automaton as an intermediate product, we consider adding support for transformation to UML StateMachines; generating an Interaction (Sequence Diagram in UML 1.x) should be feasible as well.

Future extensions of the analysis include identifying nested calls (expressed with curly braces in the manually created Pro-case in Fig. 5.13). While some actions described in a use case step complete immediately, some may have duration spanning several subsequent actions. A typical example is a request received by the SuD, followed by one or more actions taken by the SuD. The actions taken by SuD are actually performed while processing the request received; such nesting can be, in some situations, explicitly marked with a *respond clause*, e.g., step 8 (*"System responds ..."*) in Fig. 5.1. A research aim worth considering is developing rules for matching respond clauses with requests, as well as handling situations where the respond clause is omitted.

Enhancing the algorithm constructing the event tokens to create matching tokens for complementary actions in different use cases is another goal worth pursuing; here, a key step towards this goal is to define a criterion for similarity of the sets of words contained in the representative object description.

Another area to explore is simulating the execution of a use case specification, based on the behavior specification obtained. Such a simulation might be used to validate the requirement specification by generating test scenarios.

Last but not least, we also aim to perform a case study on an industrial use case requirement specification.

9.3.3. Future Work on Port State Machines

posm

To aid with integrating support for Port State Machines (PoSM) and consistency reasoning on PoSM specifications into a UML tool, we aim to formally define the compliance relation by means of the OCL language and formally capture the relation in the UML metamodel.

Moreover, we aim to propose a restricted constraint language, that would not break the regularity of the language generated by a PoSM, yet provide convenient modeling power. We consider developing a simple constraint language utilizing only the current state of the state machine (using an **`in(state)`** predicate to query orthogonal regions of the state machine); such a constraint language should fulfill the expectations: the language generated by a PoSM would remain regular, while the perceived expressive power of specifications would significantly increase.

With the aim to employ PoSMs to model use cases, our future goal is to further investigate operations for assembling behavior scattered in multiple PoSMs into a single PoSM. Currently, composition is defined only for communication languages of PoSMs, yielding the composed behavior as a language. We aim to explore composition of PoSMs at structural level, with the goal to construct a PoSM representing the composed behavior. Moreover, broadening the definition of

behavior compliance to include also state events (for entering/exiting a state) remains a challenge.

Further, as already outlined in Sect. 9.3.1, we aim to include the proposed UML extensions in a UML Profile implemented for a UML tool, providing support for PoSMs and employing the behavior compliance verifier tool [46, 77], already available for behavior protocols, to reason on compliance of PoSMs. As the verifier tool internally represents the language as a finite automaton, constructing the internal representation directly from a Port State Machine might improve efficiency of the verification. Considering that the envisioned tool operates on UML models (with the PoSM metamodel extensions), we might also integrate the tool into the Evolution and Validation Environment [79].

9.4. Conclusion

We have proposed the simple formal model Generic UC View identifying the key abstractions in use case modeling and the relations among them, thus meeting the goals (i) and (ii) outlined in Sect. 1.6. We have applied the model developed to analyze existing approaches (goal (iii)) and subsequently proposed Pro-cases (goal (iv)), a new behavior specification mechanisms addressing the issues identified in our analysis. Pro-cases support behavior assembly and composition, as well as consistency reasoning; a tool verifying the compliance relation is available [46,77]. By analyzing how textual use cases can be transformed into Pro-cases and developing a tool employing a natural language parser to automate this conversion, we have elaborated and addressed the goal (v).

Subsequently, we have extended our generic model and employed these extensions to analyze the emerging standard UML 2.0; based on the analysis, we have proposed Port State Machines, a new behavior specification mechanism based on UML 2.0 Protocol State Machines, that fits into UML 2.0 framework and address the issues we have identified in use case modeling; among others, it supports consistency reasoning. Thus, we have addressed the remaining goals (vi) and (vii) outlined for this thesis.

References

[1] Adamek, J., Plasil,F.: Behavior Protocols Capturing Errors and Updates. In *Proceedings of the 2nd International Workshop on Unanticipated Software Evolution*, ETAPS, Warsaw, 2003

[2] Adamek, J., Plasil, F.: Behavior Protocols: Tolerating Faulty Architectures and Supporting Dynamic Updates. *Technical Report 02/10*, Dept. of Computer Science, University of New Hampshire, Durham, NH, U.S.A., Oct 2002

[3] Allen, R., Garlan, D.: *A Formal Basis for Architectural Connection*, ACM Trans. Softw. Eng. Methodol. 6(3): 213-249 (1997)

[4] Amyot, D., Mussbacher, G.: *On the Extension of UML with Use Case Maps Concepts*, in Proceedings of UML 2000, York, UK, October 2-6, 2000, LNCS 1939, Springer 2000

[5] Bergstra J. A., Ponse A., Smolka S.A.: *Handbook of Process Algebra*, Elsevier 2001, ISBN 0444828303

[6] Bies, A., Ferguson, M., Katz, K., MacIntyre, R.: *Bracketing Guidelines for Treebank II Style Penn Treebank Project*, Computer and Information Science Department, University of Pennsylvania, http://www.cis.upenn.edu/~treebank/

[7] Bikel, D. M. : *Design of a Multi-lingual, Parallel-processing Statistical Parsing Engine*, in Proceedings of HLT 2002, http://www.cis.upenn.edu/~dbikel/software.html#stat-parser

[8] Bruneton, E. Coupaye, T., Stefani, J.B.: *The Fractal Component Model*, Draft 2.0-3, February 5, 2004, http://fractal.objectweb.org/specification/

[9] Buhr, R.J.A.: *Use Case Maps as Architectural Entities for Complex Systems*, Transactions on Software Engineering, IEEE, vol 24, no **12** (1998)

[10] Charniak, E.: *Statistical Techniques for Natural Language Parsing*, AI Magazine 18(4): 33-44 (1997)

[11] CLAWS part-of-speech tagger for English, http://www.comp.lancs.ac.uk/ucrel/claws/

[12] Cockburn, A.: *Harnessing Convection Currents of Information*, Invited Talk, OOPSLA 2001, October 14-18 2001, Tampa Convention Center, Tampa Bay, FL

[13] Cockburn, A.: *Writing Effective Use Cases*, Addison-Wesley Pub Co, ISBN: 0201702258, 1st edition, Jan 2000

[14] Collins, M.: *A New Statistical Parser Based on Bigram Lexical Dependencies*, ACL 1996: 184-191. 34th Annual Meeting of the Association for Computational Linguistics, 24-27 June 1996, University of California, Santa Cruz, California, USA, Proceedings. Morgan Kaufmann Publishers

[15] Collins, M.: *Head-Driven Statistical Models for Natural Language Parsing*, PhD Dissertation, Computer and Information Science Department, University of Pennsylvania, 1999, http://www.ai.mit.edu/ people/mcollins/code.html

[16] Damm, W., Harel, D.: *LSCs: Breathing Life into Message Sequence Charts.* Formal Methods in System Design 19(1): 45-80 (2001), Kluwer 2001

[17] D'Souza, D.: *Components with Catalysis*, http://www.catalysis.org/, 2001

[18] D'Souza, D. F., Wills, A. C.: *Objects, Components, and Frameworks with UML : The Catalysis(SM) Approach*, Addision-Wesley, 1998, ISBN: 0201310120

[19] Ecma International: *Common Language Infrastructure (CLI)*, 2nd edition (Dec 2002)

[20] Engels, G., Heckel, R., Küster, J. M.: *The Consistency Workbench: A Tool for Consistency Management in UML-Based Development.* in Proceedings of UML2003 - The Unified Modeling Language, Modeling Languages and Applications, San Francisco, CA, USA, Oct, 2003, LNCS 2863, pp. 356-359, Springer 2003, ISBN 3-540-20243-9

[21] Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: *Application of Linguistic Techniques for Use Case Analysis*, in Proceedings of RE 2002, pp. 157-164, Sep 9-13, 2002, Essen, Germany. IEEE Computer Society 2002

[22] Genova, G., Llorens, J., Quintana, V.: *Digging into Use Case Relationships*, UML 2002, Dresden, Germany, 2002

[23] Gervasi, V., Nuseibeh, B.: *Lightweight validation of natural language requirements*, Softw., Pract. Exper. 32(2): 113-133 (2002)

[24] Gervasi, V.: *Synthesizing ASMs from natural language requirements*, in Proc. of the 8th EUROCAST Workshop on Abstract State Machines, pages 212-215, February 2001.

[25] Gervasi, V.: *The Cico domain-based parser*. Technical Report TR-01-25, University of Pisa, Dipartimento di Informatica, November 2001.

[26] Graham, I.: *Object-Oriented Methods: Principles and Practice*, Addison-Wesley Pub Co, ISBN: 020161913X, 3rd edition December 2000

[27] Grieskamp, W., Lepper, M., Schulte, W., Tillmann, N.: *Testable Use Cases in the Abstract State Machine Language*, in Proceedings of APAQS'01, December 10 – 11, 2001, Hong Kong

[28] Harel, D. *Statecharts: A visual formalism for complex systems*, Science of Computer Programming **8** (1987), pp. 231-274, Elsevier Science Publishers B.V. (North-Holland)

[29] Hausmann, J. H., Hecke, R., Taentzer, G.: *Detection of Conflicting Functional Requirements in a Use Case-Driven Approach*, ICSE 2002, Orlando, FL, USA, May 19-25, 2002

[30] Hong, H. S., Kim, Y. G., Cha, S. D., Bae, D.-H., Ural, H.: *A test sequence selection method for statecharts*, Software Testing, Verification & Reliability vol 10 no **4** (2000), pp. 203-227

[31] Hurlbut R. R.: *A Survey of Approaches For Describing and Formalizing Use Cases*, Expertech, Ltd., Document: XPT-TR-97-03

[32] Isoda, S.: *A Critique of UML's Definition of the Use-Case Class*, in Proceedings of UML 2003, Oct 20-24, San Francisco, 2003, Springer-Verlag, LNCS 2863

[33] Jacobson, I., Christerson, M.: *A Growing Consensus on Use Cases*, JOOP 8(1): 15-19 (1995)

[34] Jacobson, I.: *Formalizing Use-Case Modeling*, JOOP 8(3): 10-14 (1995)

[35] Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Pub Co; ISBN: 0201544350; 1st edition (June 30, 1992)

[36] Kantorowitz, E., Tadmor, S.: *A Specification-Oriented Framework for Information System User Interfaces*, in Proceedigs of OOIS Workshops 2002, LNCS 2426, Springer-Verlag 2002

[37] Kristen, G.: *Object-Orientation: The Kiss Method: From Information Architecture to Information System*, ASIN: 0201422999, Addison-Wesley Pub Co, Nov 1994

[38] Kozlenkov, A., Zisman, A.: *Are their Design Specifications Consistent with our Requirements?*, Proceedings of RE 2002, pp. 145-156, Sep 9-13, 2002, Essen, Germany. IEEE Computer Society 2002, ISBN 0-7695-1465-0

[39] Kulak, D., Guiney, E.: *Use cases: requirements in context*, Addison-Wesley, Pub Co, ISBN: 0-201-65767-8, May 2000

[40] Larman, C.: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, Prentice Hall PTR, ISBN: 0130925691, 2nd ed, 2001

[41] Latella, D., Majzik, I., Massink, M.: *Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-checker*, Formal Aspects of Computing vol 11 no **6** (1999), pp. 637-664

[42] Latella, D., Massink, M.: *A Formal Testing Framework for UML Statechart Diagrams Behaviours: From Theory to Automatic Verification*, in Proceedings of HASE 2001, IEEE Computer Society (2001), pp. 11-22

[43] Leavens, G.T., Sitaraman, M. (eds.): *Foundations of Component-Based Systems*, Cambridge University Press, March 2000, ISBN: 0521771641

[44] Lee, B.-S., Bryant, B.R.: *Automated conversion from requirements documentation to an object-oriented formal specification language*, in Proceedings of SAC 2002, March 10-14, 2002, Madrid, Spain, ACM 2002

[45] Lin, D.: *MINIPAR*, http://www.cs.ualberta.ca/~lindek/

[46] Mach, M., Plasil, F.: *Addressing State Explosion in Behavior Protocol Verification*, Accepted for publication in proceedings of SNPD'04, Beijing, China, Jun 2004

[47] Magee, J., Kramer, J.: *Dynamic Structure in Software Architectures*, in Proceedings of SIGSOFT FSE 1996, San Francisco, California, USA, October 16-18, 1996. ACM SIGSOFT Software Engineering Notes 21(6), November 1996

[48] Mencl, V.: *Enhancing Component Behavior Specifications with Port State Machines*, Tech. Report No. 2003/4, Dep. of SW Engineering, Charles University, Prague, Sep 2003

[49] Mencl, V.: *Specifying Component Behavior with Port State Machines*, Accepted for publication in proceedings of Compositional Verification of UML Models workshop (Oct 21, 2003, part of UML 2003) in a volume of the Electronic Notes in Theoretical Computer Science, Elsevier Science

[50] Mencl, V.: *Autonomous Points in Component Composition*, Extended abstract of the Poster presented at OOPSLA 2001, in the Conference Companion, ACM ISBN: 1-58113-441-X, Tampa, FL, USA, Oct 2001

[51] Mencl, V., Plasil, F., Adamek, J.: *Use Cases in UML 2.0: Analyzing Support for Behavior Assembly and Composition*, work-in-progress

[52] Mencl, V.: *Converting Textual Use Cases into Behavior Specifications*, work-in-progress

[53] Mencl, V., Adamek, J., Buble, A., Hnetynka, P., Visnovsky, S.: *Enhancing EJB Component Model*, Tech. Report No. 2001/7, Dep. of SW Engineering, Charles University, Prague, Dec 2001

[54] Mencl, V.: *Managing Configuration of Update-enabled Software Components*, Tech. Report No. 2001/5, Dep. of SW Engineering, Charles University, Prague, Oct 2001

[55] Mencl, V., Hnetynka, P.: *Managing Evolution of Component Specifications using a Federation of Repositories*, Tech. Report No. 2001/2, Dep. of SW Engineering, Charles University, Prague, Jun 2001

[56] Minnen, G., Carroll J., Pearce, D.: *Applied morphological processing of English*, Natural Language Engineering, 7(3), pp. 207-223, (2001), http://www.cogs.susx.ac.uk/lab/nlp/carroll/morph.html

[57] Nierstrasz, O., Arévalo, G., Ducasse, S., Wuyts, R., Black, A., Müller, P., Zeidler, C., Genssler, T., Born, R. van den: *A Component Model for Field Devices*, IFIP/ACM Conference on Component Deployment, Berlin, Germany, June 2002

[58] Object Management Group (OMG): CORBA Component Model, v3.0, formal/02-06-65, http://www.omg.org/

[59] Object Management Group (OMG): Unified Modeling Language (UML), version 1.4, formal/2001-09-67, http://www.omg.org/uml/

[60] Object Management Group (OMG): Unified Modeling Language (UML), version 1.5, formal/2003-03-01, http://www.omg.org/uml/

[61] Object Management Group (OMG): Unified Modeling Language: Superstructure, version 2.0, Final Adopted specification, ptc/03-08-02, http://www.omg.org/uml/

[62] Object Management Group (OMG): Unified Modeling Language: Superstructure, version 2.0, Revised Final Adopted specification (FTF convenience document), ptc/04-05-02, http://www.omg.org/uml/

[63] Object Management Group (OMG): FTF Report of the UML 2.0 Superstructure Finalization Task Force, ptc/04-05-01, http://www.omg.org/uml/

[64] Odeh, M., Hauer, T., McClatchey, R., Solomonides, T.: *A Use-Case Driven Approach in Requirements Engineering : The Mammogrid Project*, Presented at the 7th IASTED Int Conf on Software Engineering Applications, Marina del Rey, USA November 2003, arXiv:cs.DB/0402008, http://arxiv.org/abs/cs.DB/0402008, Feb 2004

[65] Osborne, M., MacNish, C. K. : *Processing Natural Language Software Requirement Specifications*, Proceedings of ICRE 1996, pp. 229-237, April 15 - 18, 1996, Colorado Springs, Colorado, USA. IEEE Computer Society

[66] Plasil, F., Balek, D., Janecek, R.: *SOFA/DCUP Architecture for Component Trading and Dynamic Updating*, In *Proceedings of ICCDS '98*, Annapolis, IEEE Computer Soc. (1998)

[67] Plasil, F., Mencl, V.: *Getting "Whole Picture" Behavior in a Use Case Model*, in Proceedings of IDPT 2003, Austin, Texas, U.S.A., Dec 2003, ISSN 1090-9389

[68] Plasil, F., Mencl, V.: *Use Cases: Assembling "Whole Picture Behavior"*, TR 02/11, Dept. of Computer Science, University of New Hampshire, Durham, 2002

[69] Plasil, F., Mencl, V.: *Getting "Whole Picture" Behavior in a Use Case Model*, in Transactions of the SDPS: Journal of Integrated Design and Process Science, vol. 7, no. 4, pp. 63-79, Dec 2003, ISSN-1092-0617

[70] Plasil F., Visnovsky, S.: *Behavior Protocols for Software Components*. Transactions on Software Engineering, IEEE, vol 28, no **11** (2002)

[71] Plasil, F., Visnovsky, S., Besta, M.: *Bounding Behavior via Protocols*, Proceedings of TOOLS USA '99, Santa Barbara, CA, Aug. 1999.

[72] Ratnaparkhi, A.: *A Maximum Entropy Part-Of-Speech Tagger*, in Proceedings of the Empirical Methods in Natural Language Processing Conference, May 17-18, 1996. University of Pennsylvania, http://www.cis.upenn.edu/~adwait/statnlp.html

[73] Richards, D., Boettger, K., Aguilera, O.: *A Controlled Language to Assist Conversion of Use Case Descriptions into Concept Lattices*, Proceedings of AI 2002, Canberra, Australia, Dec 2-6, 2002, LNCS 2557 Springer 2002

[74] Rozenberg, G. (ed), Salomaa, A. (contrib.): *Handbook of Formal Languages: Word, Language, Grammar*, Springer Verlag; April 1997, ISBN: 3540604200

[75] Schneider, G., Winters, J. P.: *Applying Use Cases: A Practical Guide*, Addison-Wesley Pub Co, ISBN: 0201708531, 2nd edition, March 2001

[76] Sendall, S., Strohmeier, A.: *Using OCL and UML to Specify System Behavior*, in Clark, T., Warmer, J. (Eds.):Object Modeling with the OCL, The Rationale behind the Object Constraint Language. LNCS 2263, Springer 2002, ISBN 3-540-43169-1250-280

[77] SOFA Behavior Protocol Verifier, SOFA project, http://nenya.ms.mff.cuni.cz/projects/sofa/tools/

[78] Stevens, P.: *On Use Cases and Their Relationships in the Unified Modelling Language*, in Proceedings of FASE 2001 (part of ETAPS 2001), Genova, Italy April 2001, Springer LNCS 2029

[79] Süß, J. G., Leicher, A., Weber, H., Kutsche R.-D.: *Model-Centric Engineering with the Evolution and Validation Environment*, in Proceedings of UML 2003, Oct 20-24, San Francisco, 2003, Springer-Verlag, LNCS 2863

[80] Tenzer, J., Stevens, P.: *Modelling recursive calls with UML state diagrams*, in Proceedings of FASE 2003 (part of ETAPS 2003), Warsaw, Poland, April 7-11, 2003, LNCS 2621, Springer

[81] Tretmans, J. *Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation*. Computer Networks and ISDN Systems, vol. 29 no **1**, pp. 49-79 (1996)

[82] Uchitel, S., Kramer, J.: *A Workbench for Synthesising Behaviour Models from Scenarios*, in Proceedings of ICSE 2001, 12-19 May 2001, Toronto, Ontario, Canada

[83] Yang, C., Bryant, B. R., Burt, C. C., Raje, R. R., Olson, A. M., Auguston, M.: *Formal Methods for Quality of Service Analysis in Component-Based Distributed Computing*, in Proceedings of IDPT 2003, Dec 2003, Austin, Texas, U.S.A.

[84] Yaser A.-O., Curin, J., Jahr, M., Knight, K., Lafferty, J, Melamed, D., Och, F.-J.Purdy, D., Smith, N. A., Yarowsky, D.: Statistical Machine Translation: Final Report, Johns Hopkins University 1999 Summer Workshop (WS 99) on Language Engineering, Center for Language and Speech Processing, Baltimore, MD, USA, http://www.clsp.jhu.edu/ws99/projects/mt/toolkit/

[85] Zimmerman, M. K., Lundqvist, K., Leveson, N.: *Investigating the Readability of State-Based Formal Requirements Specification Languages*, ICSE 2002, Orlando, Florida, USA, May 2002

[86] Zuendorf, A.: *From Use Cases to Code – Rigorous Software Development with UML*, Tutorial T4, ICSE 2001: May 12-19, 2001, Toronto, Ontario, Canada

Appendix A: Frame Pro-case Expanded

To illustrate the semantics of use case expressions, we provide here a fully fledged frame Pro-case acquired by assembling the Pro-cases of system *M* using the use case expression PE^M_R derived in Sect 4.3. For the use case expression

$PE^M_R = (PC^M_1 + PC^M_4 + PC^M_5 + PC^M_6)^* \,\|\, (PC^M_7)^*$

we obtain the following frame Pro-case:

```
(
   ?sic.submitItem   { τValidateItem ; ( NULL + τPriceAssessmentAvailable ;
                       !sellernotify.putPriceAssessment + τInvalidItem ) } ;
   ( ?sic.submitPrice {
         τValidateSeller ; τVerifySellerHistory ;
         (    !tradecom.validate ;
            ( τListOffer ; !sellernotify.putAuthNr + τTradeComValidateFailed )
         + τVerifyFailed
         )
      }
   + τInvalidItem
   )
+
   ?toseller.locateOffer ; ?toseller.requestCancelOffer
   {  !sellernotify.getAck ; τValidateAck ;
      ( τInvalidAck ; !sr.provideAck ; tValidateAck ) * ;
      ( τRemoveOffer + τInvalidAck )
   }
+
   ?toseller.locateOffer ; ?toseller.requestStatus
   {  !sellernotify.getAck ; τValidateAck ;
      ( τInvalidAck ; !sr.provideAck ; tValidateAck ) * ;
      ( τGetOfferStatus + τInvalidAck )
   }
+
   ?toseller.locateOffer ; ?toseller.updateOffer
   {  !sellernotify.getAck ; τValidateAck ;
      ( τInvalidAck ; !sr.provideAck ; tValidateAck ) * ;
      ( τUpdateOffer + τInvalidAck )
   }
)* || (
   ?tobuyer.search ; ?tobuyer.narrowSearch* ; ?tobuyer.requestDetails ;
   (      ?bic.initBuyOffer { τValidateOffer + τOfferInvalid } ;
            (   ?bic.doBuyOffer { !agency.validate ; τPerformSale ;
                !sellernotif.shipItem ; τTransferPayment }
            +   τOfferInvalid
            )
   +   NULL
   )
)*
```

Appendix B: Marketplace Example: Textual Use Cases Input

We have tested the Procasor tool (described in Sect. 5.5) on a previously developed set of textual use cases. Originally, the use cases were developed in a word processing program; in this version, they were published in the appendices of [68]. We have converted them into a purely textual version, separating use case headers form use case step descriptions; also, we have created a simple domain model describing the marketplace system. The following three sections of this appendix show listings of the files containing the domain model, the use case headers and the use case step descriptions, exactly as they were used in our experiment.

B.1. Model Description

This section lists the domain model used in the Marketplace case study. In the prototype tool, the model was stored in the file `ucmp-model.txt`.

```
ACTOR SL Seller
ACTOR CL Clerk
ACTOR BU Buyer
ACTOR SU Supervisor
ACTOR CRA "Credit Verification Agency"
ACTOR CS "Computer System"
ACTOR M "Marketplace Information System"
ACTOR MKT "Marketplace"
ACTOR TC "Trade Commission"
HEADERS ucmp-uchdrs.txt
TREEFILE ucmp-lemlab.txt
DOMAINNAME item item
DOMAINNAME offer offer
DOMAINNAME pa price assessment
DOMAINNAME price price
DOMAINNAME paymentmethod payment method
```

B.2. Textual Use Cases – Headers

This section lists the headers of the use cases used in the Marketplace case study. In the prototype tool, the model was stored in the file `ucmp-uchdrs.txt`.

```
Use Case: M#1 Seller submits an offer
Scope: Marketplace
SuD: Marketplace Information System
Level: Primary Task
Primary Actor: Seller
Supporting Actor: Trade Commission

Use Case: M#2 Buyer searches for an offer
Scope: Marketplace
SuD: Marketplace Information System
Level: Primary Task
Primary Actor: Buyer
Supporting Actor:
# supporting actor missing in original data
```

```
Use Case: M#3 Buyer buys a selected item
Scope: Marketplace
SuD: Marketplace Information System
Level: Primary Task
Primary Actor: Buyer
Supporting Actor: Seller, Credit Verification Agency

Use Case: M#4 Seller cancels an offer
Scope: Marketplace
SuD: Marketplace Information System
Level: Primary Task
Primary Actor: Seller
Supporting Actor:

Use Case: M#5 Seller checks on the status of an offer
Scope: Marketplace
SuD: Marketplace Information System
Level: Primary Task
Primary Actor: Seller
Supporting Actor:

Use Case: M#6 Seller updates an offer
Scope: Marketplace
SuD: Marketplace Information System
Level: Primary Task
Primary Actor: Seller
Supporting Actor:

Use Case: M#7 Buyer makes a purchase
Scope: Marketplace
SuD: Marketplace Information System
Level: Primary Task
Primary Actor: Buyer
Supporting Actor: Seller, Credit Verification Agency


Use Case: CS#1 Clerk submits an offer on behalf of a Seller
Scope: Marketplace Information System
SuD: Computer System
Level: Primary Task
Primary Actor: Clerk
# Seller is ultimate Actor
Supporting Actor: Trade Commission, Supervisor, Seller
# Supporting Actor accidentally omitted
# Seller is also a Supporting Actor - omitted in original data

Use Case: CS#2 Buyer searches for an offer
Scope: Marketplace Information System
SuD: Computer System
Level: Primary Task
Primary Actor: Buyer
Supporting Actor:
# supporting actor missing in original data

Use Case: CS#3 Clerk buys a selected item on behalf of a Buyer
Scope: Marketplace Information System
SuD: Computer System
Level: Primary Task
Primary Actor: Clerk
# ultimate actor Buyer
Supporting Actor: Seller, Credit Verification Agency, Buyer
# Buyer is also a Supporting Actor - omitted in original data
```

```
Use Case: CS#4 Seller cancels an offer
Scope: Marketplace Information System
SuD: Computer System
Level: Primary Task
Primary Actor: Seller
Supporting Actor:

Use Case: CS#5 Seller checks on the status of an offer
Scope: Marketplace Information System
SuD: Computer System
Level: Primary Task
Primary Actor: Seller
Supporting Actor:

Use Case: CS#6 Seller updates an offer
Scope: Marketplace Information System
SuD: Computer System
Level: Primary Task
Primary Actor: Seller
Supporting Actor:

Use Case: CS#7 Buyer makes a purchase
Scope: Marketplace Information System
SuD: Computer System
Level: Primary Task
Primary Actor: Buyer
Supporting Actor: Clerk, Seller, Credit Verification Agency

Use Case: CS#8 Supervisor makes an internal audit
Scope: Marketplace Information System
SuD: Computer System
Level: Primary Task
Primary Actor: Supervisor
Supporting Actor:

Use Case: CL#1 Seller to Clerk
Scope: Marketplace Information System
SuD: Clerk
Level: Primary Task
Primary Actor: Seller
Supporting Actor: Computer System

Use Case: CL#2 Buyer to Clerk
Scope: Marketplace Information System
SuD: Clerk
Level: Primary Task
Primary Actor: Buyer
Supporting Actor: Computer System, Seller
# supporting actor Seller omitted in original data

Use Case: SU#1 Supervisor validates a seller
Scope: Marketplace Information System
SuD: Supervisor
Level: Primary Task
Primary Actor: Computer System
Supporting Actor:
```

Use Case: SU#2 Supervisor performs internal audit
Scope: Marketplace Information System
SuD: Supervisor
Level: Primary Task
Primary Actor: Computer System
Supporting Actor:

B.3. Textual Use Cases – Step Descriptions

This section lists the descriptions of steps and extension conditions of the use cases used in the Marketplace case study. In the prototype tool, the model was stored in the file `ucmp-labeled.txt`. Each line of the file describes a single step and/or extension condition of a use case; the line consists of a label composed of identification of the use case and of a label of the step and/or extension, followed by the textual description.

```
M1-1.     Seller submits item description.
M1-2.     System validates the description.
M1-3.     Seller adjusts/enters price and enters contact and
          billing information.
M1-4.     System validates the seller's contact information.
M1-5.     System verifies the seller's history.
M1-6.     System validates the whole offer with the Trade
          Commission
M1-7.     System lists the offer in published offers.
M1-8.     System responds with an uniquely identified
          authorization number.
M1-2a     Item not valid
M1-2a1    Use case aborts
M1-5a     Seller's history inappropriate
M1-5a1    Use case aborts
M1-6a     Trade commission rejects the offer
M1-6a1    Use case aborts
M1-2b     Price assessment available
M1-2b1    System provides the seller with a price assessment.
M2-1.     Buyer enters basic search criteria.
M2-2.     System responds with the list of matches.
M2-3.     Buyer requests the complete listing of a selected offer.
M2-4.     System responds with the requested information.
M2-2a     No matches found
M2-2a1    Use case aborted
M2-2b     The amount of matches is too high
M2-2b1    Buyer narrows the search results with additional
          criteria
M2-2b2    Resume with step 2
M3-1.     Buyer chooses to accept a selected offer.
M3-2.     System validates the offer.
M3-3.     User enters billing information, select a payment method
          and provides the payment details.
M3-4.     System validates the buyer's information with the Credit
          Verification Agency.
M3-5.     System performs the sale.
M3-6.     System informs the seller that the offer has been
          accepted and provides the shipping information.
M3-7.     System transfers the payment to the seller's account.
M3-8.     System responds to the buyer with an uniquely identified
          authorization number.
M3-2a     Offer is not valid
M3-2a1    Use case is aborted
M4-1.     Seller locates a previously submitted offer.
M4-2.     Seller requests the system to cancel the offer.
M4-3.     System responds with a request for the seller to prove
          identity.
M4-4.     Seller responds with the authorization number obtained
          when the offer was submitted.
M4-5.     System validates the request and seller's identity..
M4-6.     System removes the offer.
M4-4a     Seller cannot provide the authorization number
```

M4-4a1 Use case is aborted
M4-5a Authorization number is not valid
M4-5a1 Retry with step 3
M5-1. Seller locates a previously submitted offer.
M5-2. Seller requests the system to provide status of the offer.
M5-3. System responds with a request for the seller to prove identity.
M5-4. Seller responds with the authorization number returned when the offer was submitted.
M5-5. System validates the request and seller's identity..
M5-6. System returns the status of the offer.
M5-4a Seller cannot provide the authorization number
M5-4a1 Use case is aborted
M5-5a Authorization number is not valid
M5-5a1 Retry with step 3
M6-1. Seller locates a previously submitted offer.
M6-2. Seller requests the system to update the offer, providing new details (e.g., price).
M6-3. System responds with a request for the seller to prove identity.
M6-4. Seller responds with the authorization number returned when the offer was submitted.
M6-5. System validates the request and seller's identity..
M6-6. System updates the offer.
M6-4a Seller cannot provide the authorization number
M6-4a1 Use case is aborted
M6-5a Authorization number is not valid
M6-5a1 Retry with step 3
M7-1. Buyer searches for an offer (#2)
M7-2. Buyer buys a selected item (#3)
M7-1a The Buyer did not find any matching offer
M7-1a1 Use case aborted
M7-1b The Buyer decides not to accept the offer.
M7-1b1 Use case ends here.
CS1-1. Clerk submits information describing an item
CS1-2. System validates the description.
CS1-3. Clerk adjusts/enters price and enters seller's contact and billing information.
CS1-4. System validates the seller's contact information.
CS1-5. System asks the Supervisor to validate the seller.
CS1-6. Supervisor permits the seller to operate on the marketplace.
CS1-7. System validates the whole offer with the Trade Commission
CS1-8. System lists the offer in published offers.
CS1-9. System responds with an uniquely identified acknowledgment.
CS1-2a Validation performed by the system fails
CS1-2a1 Use case aborted
CS1-2b Price assessment available
CS1-2b1 System provides the seller with a price assessment.
CS1-7a Trade commission rejects the offer
CS1-7a1 Use case aborted
CS2-1. Buyer enters basic search criteria.
CS2-2. System responds with the list of matches.
CS2-3. Buyer requests the complete listing of a selected offer.
CS2-4. System responds with the requested information.
CS2-2a No matches found
CS2-2a1 Use case aborted
CS2-2b The amount of matches is too high

CS2-2b1 Buyer narrows the search results with additional criteria
CS2-2b2 Resume with step 2
CS3-1. Clerk is contacted by a buyer who has decided to accept a selected offer.
CS3-2. System validates the offer.
CS3-3. System requests billing and shipping information, payment method and payment detail information.
CS3-4. Clerk enters billing information, select a payment method and provides the necessary details.
CS3-5. System validates this information with a Credit Verification Agency.
CS3-6. System performs the trade.
CS3-7. System informs the seller that the offer has been accepted and provides the shipping information.
CS3-8. System transfers the payment to the seller's account.
CS3-9. System responds to the buyer with an uniquely identified authorization number.
CS3-2a Offer is not valid
CS3-2a1 Use case is aborted
CS4-1. Seller locates a previously submitted offer.
CS4-2. Seller requests the system to cancel the offer.
CS4-3. System responds with a request for the seller to prove identity.
CS4-4. Seller responds with the authorization number returned when the offer was submitted.
CS4-5. System validates the request and seller's identity..
CS4-6. System removes the offer.
CS4-4a Seller cannot provide the authorization number
CS4-4a1 Use case is aborted
CS4-5a Authorization number is not valid
CS4-5a1 Retry with step 3
CS5-1. Seller locates a previously submitted offer.
CS5-2. Seller requests the system to provide status of the offer.
CS5-3. System responds with a request for the seller to prove identity.
CS5-4. Seller responds with the authorization number returned when the offer was submitted.
CS5-5. System validates the request and seller's identity..
CS5-6. System returns the status of the offer.
CS5-4a Seller cannot provide the authorization number
CS5-4a1 Use case is aborted
CS5-5a Authorization number is not valid
CS5-5a1 Retry with step 3
CS6-1. Seller locates a previously submitted offer.
CS6-2. Seller requests the system to update the offer, providing new details (e.g., price).
CS6-3. System responds with a request for the seller to prove identity.
CS6-4. Seller responds with the authorization number returned when the offer was submitted.
CS6-5. System validates the request and seller's identity..
CS6-6. System updates the offer.
CS6-4a Seller cannot provide the authorization number
CS6-4a1 Use case is aborted
CS6-5a Authorization number is not valid
CS6-5a1 Retry with step 3
CS7-1. Buyer searches for an offer (#2/F)
CS7-2. Buyer contacts a clerk to buys the selected item (#3/P)
CS7-1a The Buyer did not find any matching offer
CS7-1a1 Use case aborted

CS7-1b The Buyer decides not to accept the offer.
CS7-1b1 Use case ends here.
CS8-1. Supervisor searches the database of offers for sensitive keywords in item description
CS8-2. Supervisor displays the description of the item.
CS8-3. Supervisor removes the item from the database of currently visible offers.
CS8-1a Supervisor did not find any match
CS8-1a1 Use case terminates
CS8-2a Supervisor did not find any offending items.
CS8-2a1 Use case terminates
CS8-2b Supervisor requests details of another item
CS8-2b1 Repeat step 2
CL1-1. Seller submits item description to the clerk.
CL1-2. Clerk submits the description to the system.
CL1-3. Clerk reports the system response to the seller.
CL1-4. Seller submits the price, billing and contact information to the clerk.
CL1-5. Clerk enters the price, billing and contact information to the system.
CL1-6. Clerk reports the system response to the seller.
CL1-2a Validation performed by the system fails
CL1-2a1 Use case aborted
CL2-1. Buyer submits to the clerk a reference to a selected offer.
CL2-2. Clerk submits the reference to the system.
CL2-3. Clerk reports the system response to the seller and requests billing and shipping information, payment method and payment details.
CL2-4. Buyer submits to the clerk the requested billing and shipping information, payment method and payment details.
CL2-5. Clerk enters the billing and shipping information, payment method and payment details.
CL2-6. Clerk reports the system response (with the unique acknowledgment) to the buyer.
CL2-3a System failed to validate the offer
CL2-3a1 Use case aborted
SU1-1. Computer system asks the supervisor to decide on permitting a seller to operate on the marketplace.
SU1-2. System validates the seller and signals the system to permit the seller to operate.
SU2-1. Supervisor requests the computer system to search the database of offers for sensitive keywords in item description
SU2-2. Supervisor requests from the computer system detailed descriptions of an item found.
SU2-3. Supervisor requests the computer system to remove the item from the database of currently visible offers.
SU2-1a No matching item found
SU2-1a1 Use case terminates
SU2-2a The item does is not an offending item.
SU2-2a1 Use case terminates
SU2-2b Supervisor requests details of another item
SU2-2b1 Repeat step 2

Appendix C: Intermediate Data: Annotated Parse Trees

This appendix contains the parse trees obtained for the use case step descriptions listed in Appendix B.3 by applying the linguistic tools as described in Sect. 5.5. For each line of the file `ucmp-labeled.txt` listed in Appendix B.3, a parse tree has been created; each line of the file ucmp-lemlab.txt listed here contains the label identical to the label used in Appendix B.3, followed by the parse tree; in the notation, parentheses are used to express nesting. For each leaf of the tree, the word, its POS-tag and lemma are shown, separated with slashes (/); for each non-leaf node, the tree s step la lists the descriptions of steps and extension conditions of the use cases used in Appendix B.3 the Marketplace case study. In the prototype tool, the model was stored in the file `ucmp-labeled.txt`. Each line of the file describes a single step and/or extension condition of a use case; the line consists of a label composed of identification of the use case and of a label of the step and/or extension, followed by the textual description.

Parse trees of Textual Use Cases, annotated with lemmas: file ucmp-lemlab.txt

```
M1-1. (TOP~submits~1~1 (S~submits~2~2 (NPB~Seller~1~1
Seller/NNP/Seller ) (VP~submits~2~1 submits/VBZ/submit
(NPB~description~2~2 item/NN/item description/NN/description
./PUNC./. ) ) ) )

M1-2. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System ) (VP~validates~2~1 validates/VBZ/validate
(NPB~description~2~2 the/DT/the description/NN/description
./PUNC./. ) ) ) )

M1-3. (TOP~adjusts~1~1 (S~adjusts~2~2 (NPB~Seller~1~1
Seller/NNP/Seller ) (VP~adjusts~3~1 (VP~adjusts~2~1
adjusts/VBZ/adjust (SBAR~enters~1~1 (S~enters~2~2 (NPB~/~1~1
//DT// ) (VP~enters~2~1 enters/VBZ/enter (NPB~price~1~1
price/NN/price ) ) ) ) ) and/CC/and (VP~enters~2~1
enters/VBZ/enter (NPB~information~4~4 contact/NN/contact
and/CC/and billing/NN/billing information/NN/information ./PUNC./.
) ) ) ) )

M1-4. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System ) (VP~validates~2~1 validates/VBZ/validate
(SBAR~s~1~1 (S~s~2~2 (NPB~'~3~3 the/DT/the seller/NN/seller
'/POS/' ) (VP~s~2~1 s/VBZ/s (NPB~information~2~2
contact/NN/contact information/NN/information ./PUNC./. ) ) ) ) )
) )

M1-5. (TOP~verifies~1~1 (S~verifies~2~2 (NPB~System~1~1
System/NNP/System ) (VP~verifies~2~1 verifies/VBZ/verify
(SBAR~s~1~1 (S~s~2~2 (NPB~'~3~3 the/DT/the seller/NN/seller
'/POS/' ) (VP~s~2~1 s/VBZ/s (NPB~history~1~1 history/NN/history
./PUNC./. ) ) ) ) ) ) )

M1-6. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System ) (VP~validates~3~1 validates/VBZ/validate
(NPB~offer~3~3 the/DT/the whole/JJ/whole offer/NN/offer )
(PP~with~2~1 with/IN/with (NPB~Commission~3~3 the/DT/the
Trade/NNP/Trade Commission/NNP/Commission ) ) ) ) )

M1-7. (TOP~lists~1~1 (S~lists~2~2 (NPB~System~1~1
System/NNP/System ) (VP~lists~3~1 lists/VBZ/list (NPB~offer~2~2
```

the/DT/the offer/NN/offer) (PP~in~2~1 in/IN/in (NPB~offers~2~2
published/VBN/publish offers/NNS/offer ./PUNC./.)))))

M1-8. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NPB~number~4~4 an/DT/an
(ADJP~identified~2~2 uniquely/RB/uniquely identified/VBN/identify
) authorization/NN/authorization number/NN/number ./PUNC./.)))
))

M1-2a (TOP~Item~1~1 (NP~Item~2~1 (NPB~Item~1~1 Item/NNP/Item)
(ADJP~valid~2~2 not/RB/not valid/JJ/valid)))

M1-2a1 (TOP~aborts~1~1 (S~aborts~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~aborts~1~1 aborts/VBZ/abort)))

M1-5a (TOP~s~1~1 (S~s~2~2 (NPB~'~2~2 Seller/NNP/Seller '/POS/')
(VP~s~3~1 s/VBZ/s (NPB~history~1~1 history/NN/history)
(ADJP~inappropriate~1~1 inappropriate/JJ/inappropriate))))

M1-5a1 (TOP~aborts~1~1 (S~aborts~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~aborts~1~1 aborts/VBZ/abort)))

M1-6a (TOP~rejects~1~1 (S~rejects~2~2 (NPB~commission~2~2
Trade/NNP/Trade commission/NN/commission) (VP~rejects~2~1
rejects/VBZ/reject (NPB~offer~2~2 the/DT/the offer/NN/offer)))
)

M1-6a1 (TOP~aborts~1~1 (S~aborts~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~aborts~1~1 aborts/VBZ/abort)))

M1-2b (TOP~assessment~1~1 (NP~assessment~2~1 (NPB~assessment~2~2
Price/NNP/Price assessment/NN/assessment) (ADJP~available~1~1
available/JJ/available)))

M1-2b1 (TOP~provides~1~1 (S~provides~2~2 (NPB~System~1~1
System/NNP/System) (VP~provides~3~1 provides/VBZ/provide
(NPB~seller~2~2 the/DT/the seller/NN/seller) (PP~with~2~1
with/IN/with (NPB~assessment~3~3 a/DT/a price/NN/price
assessment/NN/assessment ./PUNC./.)))))

M2-1. (TOP~enters~1~1 (S~enters~2~2 (NPB~Buyer~1~1 Buyer/NNP/Buyer
) (VP~enters~2~1 enters/VBZ/enter (NPB~criteria~3~3 basic/JJ/basic
search/NN/search criteria/NNS/criterion ./PUNC./.))))

M2-2. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NP~list~2~1 (NPB~list~2~2 the/DT/the
list/NN/list) (PP~of~2~1 of/IN/of (NPB~matches~1~1
matches/VBZ/match ./PUNC./.)))))))

M2-3. (TOP~requests~1~1 (S~requests~2~2 (NPB~Buyer~1~1
Buyer/NNP/Buyer) (VP~requests~2~1 requests/NNS/request
(NP~listing~2~1 (NPB~listing~3~3 the/DT/the complete/JJ/complete
listing/NN/listing) (PP~of~2~1 of/IN/of (NPB~offer~3~3 a/DT/a
selected/VBN/select offer/NN/offer ./PUNC./.))))))

M2-4. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NPB~information~3~3 the/DT/the
requested/VBN/request information/NN/information ./PUNC./.))))
)

M2-2a (TOP~matches~1~1 (S~matches~2~2 (NPB~No~1~1 No/DT/No)
(VP~matches~2~1 matches/VBZ/match (VP~found~1~1 found/VBN/find))
))

M2-2a1 (TOP~aborted~1~1 (S~aborted~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~aborted~1~1 aborted/VBD/abort)))

M2-2b (TOP~is~1~1 (S~is~2~2 (NP~amount~2~1 (NPB~amount~2~2
The/DT/The amount/NN/amount) (PP~of~2~1 of/IN/of (NPB~matches~1~1
matches/VBZ/match))) (VP~is~2~1 is/VBZ/be (ADJP~high~2~2
too/RB/too high/JJ/high))))

M2-2b1 (TOP~narrows~1~1 (S~narrows~2~2 (NPB~Buyer~1~1
Buyer/NNP/Buyer) (VP~narrows~2~1 narrows/VBZ/narrow
(NP~results~2~1 (NPB~results~3~3 the/DT/the search/NN/search
results/NNS/result) (PP~with~2~1 with/IN/with (NPB~criteria~2~2
additional/JJ/additional criteria/NNS/criterion))))))

M2-2b2 (TOP~Resume~1~1 (S~Resume~1~1 (VP~Resume~2~1
Resume/VB/Resume (PP~with~2~1 with/IN/with (NPB~step~2~1
step/NN/step 2/CD/2)))))

M3-1. (TOP~chooses~1~1 (S~chooses~2~2 (NPB~Buyer~1~1
Buyer/NNP/Buyer) (VP~chooses~2~1 chooses/VBZ/choose (S~to~1~1
(VP~to~2~1 to/TO/to (VP~accept~2~1 accept/VB/accept (NPB~offer~3~3
a/DT/a selected/VBN/select offer/NN/offer ./PUNC./.)))))))

M3-2. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~2~1 validates/VBZ/validate
(NPB~offer~2~2 the/DT/the offer/NN/offer ./PUNC./.))))

M3-3. (TOP~enters~1~1 (S~enters~2~2 (NPB~User~1~1 User/NNP/User)
(VP~enters~3~1 enters/VBZ/enter (NPB~information~2~2
billing/NN/billing information/NN/information ,/PUNC,/,)
(S~select~1~1 (VP~select~3~1 (VP~select~2~1 select/VB/select
(NPB~method~3~3 a/DT/a payment/NN/payment method/NN/method))
and/CC/and (VP~provides~2~1 provides/VBZ/provide (NPB~details~3~3
the/DT/the payment/NN/payment details/NNS/detail ./PUNC./.))))
)))

M3-4. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~2~1 validates/VBZ/validate
(SBAR~s~1~1 (S~s~2~2 (NPB~'~3~3 the/DT/the buyer/NN/buyer '/POS/'
) (VP~s~3~1 s/VBZ/s (NPB~information~1~1
information/NN/information) (PP~with~2~1 with/IN/with
(NPB~Agency~4~4 the/DT/the Credit/NNP/Credit
Verification/NNP/Verification Agency/NNP/Agency ./PUNC./.))))
))))

M3-5. (TOP~performs~1~1 (S~performs~2~2 (NPB~System~1~1
System/NNP/System) (VP~performs~2~1 performs/VBZ/perform
(NPB~sale~2~2 the/DT/the sale/NN/sale ./PUNC./.))))

M3-6. (TOP~informs~1~1 (S~informs~2~2 (NPB~System~1~1
System/NNP/System) (VP~informs~2~1 informs/VBZ/inform
(NP~seller~2~1 (NPB~seller~2~2 the/DT/the seller/NN/seller)
(SBAR~that~2~1 (WHNP~that~1~1 that/WDT/that) (S~has~2~2
(NPB~offer~2~2 the/DT/the offer/NN/offer) (VP~has~3~1 (VP~has~2~1
has/VBZ/have (VP~been~2~1 been/VBN/be (VP~accepted~1~1
accepted/VBN/accept))) and/CC/and (VP~provides~2~1
provides/VBZ/provide (NPB~information~3~3 the/DT/the

shipping/NN/shipping information/NN/information ./PUNC./.))))
)))))

M3-7. (TOP~transfers~1~1 (S~transfers~2~2 (NPB~System~1~1
System/NNP/System) (VP~transfers~2~1 transfers/VBZ/transfer
(NP~account~4~4 (NP~payment~2~1 (NPB~payment~2~2 the/DT/the
payment/NN/payment) (PP~to~2~1 to/TO/to (NPB~seller~2~2
the/DT/the seller/NN/seller))) '/POS/' s/VBZ/s
account/NN/account ./PUNC./.))))

M3-8. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~to~2~1 to/TO/to (NP~buyer~2~1 (NPB~buyer~2~2 the/DT/the
buyer/NN/buyer) (PP~with~2~1 with/IN/with (NPB~number~4~4
an/DT/an (ADJP~identified~2~2 uniquely/RB/uniquely
identified/VBN/identify) authorization/NN/authorization
number/NN/number ./PUNC./.)))))))

M3-2a (TOP~is~1~1 (S~is~2~2 (NPB~Offer~1~1 Offer/NNP/Offer)
(VP~is~2~1 is/VBZ/be (ADJP~valid~2~2 not/RB/not valid/JJ/valid))
))

M3-2a1 (TOP~is~1~1 (S~is~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~is~2~1 is/VBZ/be (ADJP~aborted~1~1
aborted/JJ/aborted))))

M4-1. (TOP~locates~1~1 (S~locates~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~locates~2~1 locates/VBZ/locate
(NPB~offer~4~4 a/DT/a previously/RB/previously
submitted/VBN/submit offer/NN/offer ./PUNC./.))))

M4-2. (TOP~requests~1~1 (NP~requests~2~1 (NPB~requests~2~2
Seller/NNP/Seller requests/NNS/request) (S~to~2~2 (NPB~system~2~2
the/DT/the system/NN/system) (VP~to~2~1 to/TO/to (VP~cancel~2~1
cancel/VB/cancel (NPB~offer~2~2 the/DT/the offer/NN/offer
./PUNC./.))))))

M4-3. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NP~request~3~1 (NPB~request~2~2 a/DT/a
request/NN/request) (PP~for~2~1 for/IN/for (NPB~seller~2~2
the/DT/the seller/NN/seller)) (S~to~1~1 (VP~to~2~1 to/TO/to
(VP~prove~2~1 prove/VB/prove (NPB~identity~1~1
identity/NN/identity ./PUNC./.))))))))

M4-4. (TOP~responds~1~1 (S~responds~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NP~number~2~1 (NPB~number~3~3
the/DT/the authorization/NN/authorization number/NN/number)
(VP~obtained~2~1 obtained/VBN/obtain (SBAR~when~2~1
(WHADVP~when~1~1 when/WRB/when) (S~was~2~2 (NPB~offer~2~2
the/DT/the offer/NN/offer) (VP~was~2~1 was/VBD/be
(VP~submitted.~1~1 submitted./VBN/submitted.)))))))))

M4-5. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~2~1 validates/VBZ/validate
(SBAR~s~1~1 (S~s~2~2 (NPB~'~5~5 the/DT/the request/NN/request
and/CC/and seller/NN/seller '/POS/') (VP~s~2~1 s/VBZ/s
(NPB~identity~1~1 identity/NN/identity ../PUNC./..))))))

M4-6. (TOP~removes~1~1 (S~removes~2~2 (NPB~System~1~1
System/NNP/System) (VP~removes~2~1 removes/VBZ/remove
(NPB~offer~2~2 the/DT/the offer/NN/offer ./PUNC./.))))

M4-4a (TOP~provide~1~1 (S~provide~2~2 (NPB~cannot~2~2
Seller/NNP/Seller cannot/NN/cannot) (VP~provide~2~1
provide/VBP/provide (NPB~number~3~3 the/DT/the
authorization/NN/authorization number/NN/number))))

M4-4a1 (TOP~is~1~1 (S~is~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~is~2~1 is/VBZ/be (ADJP~aborted~1~1
aborted/JJ/aborted))))

M4-5a (TOP~is~1~1 (S~is~2~2 (NPB~number~2~2
Authorization/NNP/Authorization number/NN/number) (VP~is~2~1
is/VBZ/be (ADJP~valid~2~2 not/RB/not valid/JJ/valid))))

M4-5a1 (TOP~Retry~1~1 (NP~Retry~2~1 (NPB~Retry~1~1 Retry/NN/Retry
) (PP~with~2~1 with/IN/with (NPB~step~2~1 step/NN/step 3/CD/3))
))

M5-1. (TOP~locates~1~1 (S~locates~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~locates~2~1 locates/VBZ/locate
(NPB~offer~4~4 a/DT/a previously/RB/previously
submitted/VBN/submit offer/NN/offer ./PUNC./.))))

M5-2. (TOP~requests~1~1 (NP~requests~2~1 (NPB~requests~2~2
Seller/NNP/Seller requests/NNS/request) (S~to~2~2 (NPB~system~2~2
the/DT/the system/NN/system) (VP~to~2~1 to/TO/to (VP~provide~2~1
provide/VB/provide (NP~status~2~1 (NPB~status~1~1 status/NN/status
) (PP~of~2~1 of/IN/of (NPB~offer~2~2 the/DT/the offer/NN/offer
./PUNC./.))))))))

M5-3. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NP~request~3~1 (NPB~request~2~2 a/DT/a
request/NN/request) (PP~for~2~1 for/IN/for (NPB~seller~2~2
the/DT/the seller/NN/seller)) (S~to~1~1 (VP~to~2~1 to/TO/to
(VP~prove~2~1 prove/VB/prove (NPB~identity~1~1
identity/NN/identity ./PUNC./.))))))))

M5-4. (TOP~responds~1~1 (S~responds~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~responds~3~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NPB~number~3~3 the/DT/the
authorization/NN/authorization number/NN/number))
(VP~returned~2~1 returned/VBN/return (SBAR~when~2~1
(WHADVP~when~1~1 when/WRB/when) (S~was~2~2 (NPB~offer~2~2
the/DT/the offer/NN/offer) (VP~was~2~1 was/VBD/be
(VP~submitted.~1~1 submitted./VBN/submitted.))))))))

M5-5. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~2~1 validates/VBZ/validate
(SBAR~s~1~1 (S~s~2~2 (NPB~'~5~5 the/DT/the request/NN/request
and/CC/and seller/NN/seller '/POS/') (VP~s~2~1 s/VBZ/s
(NPB~identity~1~1 identity/NN/identity ../PUNC./..)))))))

M5-6. (TOP~returns~1~1 (S~returns~2~2 (NPB~System~1~1
System/NNP/System) (VP~returns~2~1 returns/VBZ/return
(NP~status~2~1 (NPB~status~2~2 the/DT/the status/NN/status)
(PP~of~2~1 of/IN/of (NPB~offer~2~2 the/DT/the offer/NN/offer
./PUNC./.))))))

M5-4a (TOP~provide~1~1 (S~provide~2~2 (NPB~cannot~2~2
Seller/NNP/Seller cannot/NN/cannot) (VP~provide~2~1
provide/VBP/provide (NPB~number~3~3 the/DT/the
authorization/NN/authorization number/NN/number))))

M5-4a1 (TOP~is~1~1 (S~is~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~is~2~1 is/VBZ/be (ADJP~aborted~1~1
aborted/JJ/aborted))))

M5-5a (TOP~is~1~1 (S~is~2~2 (NPB~number~2~2
Authorization/NNP/Authorization number/NN/number) (VP~is~2~1
is/VBZ/be (ADJP~valid~2~2 not/RB/not valid/JJ/valid))))

M5-5a1 (TOP~Retry~1~1 (NP~Retry~2~1 (NPB~Retry~1~1 Retry/NN/Retry
) (PP~with~2~1 with/IN/with (NPB~step~2~1 step/NN/step 3/CD/3))
))

M6-1. (TOP~locates~1~1 (S~locates~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~locates~2~1 locates/VBZ/locate
(NPB~offer~4~4 a/DT/a previously/RB/previously
submitted/VBN/submit offer/NN/offer ./PUNC./.))))

M6-2. (TOP~requests~1~1 (NP~requests~2~1 (NPB~requests~2~2
Seller/NNP/Seller requests/NNS/request) (S~to~2~2 (NPB~system~2~2
the/DT/the system/NN/system) (VP~to~2~1 to/TO/to (VP~update~2~1
update/VB/update (NP~offer~2~1 (NPB~offer~2~2 the/DT/the
offer/NN/offer ,/PUNC,/,) (VP~providing~2~1 providing/VBG/provide
(NP~details~2~1 (NPB~details~2~2 new/JJ/new details/NNS/detail)
(PRN~-LRB-~3~1 -LRB-/-LRB-/-LRB- (NP~g~2~1 (NPB~g~2~2 e/NNP/e
./PUNC./. g/NNP/g ./PUNC./. ,/PUNC,/,) (NPB~price~1~1
price/NN/price)) -RRB-/-RRB-/-RRB- ./PUNC./.))))))))

M6-3. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NP~request~3~1 (NPB~request~2~2 a/DT/a
request/NN/request) (PP~for~2~1 for/IN/for (NPB~seller~2~2
the/DT/the seller/NN/seller)) (S~to~1~1 (VP~to~2~1 to/TO/to
(VP~prove~2~1 prove/VB/prove (NPB~identity~1~1
identity/NN/identity ./PUNC./.))))))))

M6-4. (TOP~responds~1~1 (S~responds~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~responds~3~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NPB~number~3~3 the/DT/the
authorization/NN/authorization number/NN/number))
(VP~returned~2~1 returned/VBN/return (SBAR~when~2~1
(WHADVP~when~1~1 when/WRB/when) (S~was~2~2 (NPB~offer~2~2
the/DT/the offer/NN/offer) (VP~was~2~1 was/VBD/be
(VP~submitted.~1~1 submitted./VBN/submitted.)))))))

M6-5. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~2~1 validates/VBZ/validate
(SBAR~s~1~1 (S~s~2~2 (NPB~'~5~5 the/DT/the request/NN/request
and/CC/and seller/NN/seller '/POS/') (VP~s~2~1 s/VBZ/s
(NPB~identity~1~1 identity/NN/identity ../PUNC./..))))))

M6-6. (TOP~updates~1~1 (S~updates~2~2 (NPB~System~1~1
System/NNP/System) (VP~updates~2~1 updates/VBZ/update
(NPB~offer~2~2 the/DT/the offer/NN/offer ./PUNC./.))))

M6-4a (TOP~provide~1~1 (S~provide~2~2 (NPB~cannot~2~2
Seller/NNP/Seller cannot/NN/cannot) (VP~provide~2~1

```
provide/VBP/provide (NPB~number~3~3 the/DT/the
authorization/NN/authorization number/NN/number ) ) ) )

M6-4a1 (TOP~is~1~1 (S~is~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case ) (VP~is~2~1 is/VBZ/be (ADJP~aborted~1~1
aborted/JJ/aborted ) ) ) )

M6-5a (TOP~is~1~1 (S~is~2~2 (NPB~number~2~2
Authorization/NNP/Authorization number/NN/number ) (VP~is~2~1
is/VBZ/be (ADJP~valid~2~2 not/RB/not valid/JJ/valid ) ) ) )

M6-5a1 (TOP~Retry~1~1 (NP~Retry~2~1 (NPB~Retry~1~1 Retry/NN/Retry
) (PP~with~2~1 with/IN/with (NPB~step~2~1 step/NN/step 3/CD/3 ) )
) )

M7-1. (TOP~searches~1~1 (S~searches~2~2 (NPB~Buyer~1~1
Buyer/NNP/Buyer ) (VP~searches~2~1 searches/VBZ/search (PP~for~2~1
for/IN/for (NPB~offer~2~2 an/DT/an offer/NN/offer ) ) ) ) )

M7-2. (TOP~buys~1~1 (S~buys~2~2 (NPB~Buyer~1~1 Buyer/NNP/Buyer )
(VP~buys~2~1 buys/VBZ/buy (NPB~item~3~3 a/DT/a selected/VBN/select
item/NN/item ) ) ) )

M7-1a (TOP~did~1~1 (S~did~2~2 (NPB~Buyer~2~2 The/DT/The
Buyer/NNP/Buyer ) (VP~did~3~1 did/VBD/do not/RB/not (VP~find~2~1
find/VB/find (NPB~offer~3~3 any/DT/any matching/NN/matching
offer/NN/offer ) ) ) ) )

M7-1a1 (TOP~aborted~1~1 (S~aborted~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case ) (VP~aborted~1~1 aborted/VBD/abort ) ) )

M7-1b (TOP~decides~1~1 (S~decides~2~2 (NPB~Buyer~2~2 The/DT/The
Buyer/NNP/Buyer ) (VP~decides~2~1 decides/VBZ/decide (S~to~2~2
not/RB/not (VP~to~2~1 to/TO/to (VP~accept~2~1 accept/VB/accept
(NPB~offer~2~2 the/DT/the offer/NN/offer ./PUNC./. ) ) ) ) ) ) )

M7-1b1 (TOP~ends~1~1 (S~ends~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case ) (VP~ends~2~1 ends/VBZ/end (ADVP~here~1~1
here/RB/here ./PUNC./. ) ) ) )

CS1-1. (TOP~submits~1~1 (S~submits~2~2 (NPB~Clerk~1~1
Clerk/NNP/Clerk ) (VP~submits~2~1 submits/VBZ/submit
(NP~information~2~1 (NPB~information~1~1
information/NN/information ) (VP~describing~2~1
describing/VBG/describe (NPB~item~2~2 an/DT/an item/NN/item ) ) )
) ) )

CS1-2. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System ) (VP~validates~2~1 validates/VBZ/validate
(NPB~description~2~2 the/DT/the description/NN/description
./PUNC./. ) ) ) )

CS1-3. (TOP~adjusts~1~1 (S~adjusts~2~2 (NPB~Clerk~1~1
Clerk/NNP/Clerk ) (VP~adjusts~2~1 adjusts/VBZ/adjust
(SBAR~enters~1~1 (S~enters~2~2 (NPB~/~1~1 //DT// ) (VP~enters~2~1
enters/VBZ/enter (NP~price~5~1 (NPB~price~1~1 price/NN/price )
and/CC/and (NP~contact~5~5 enters/VBZ/enter seller/NN/seller
'/POS/' s/VBZ/s contact/NN/contact ) and/CC/and
(NPB~information~2~2 billing/NN/billing information/NN/information
./PUNC./. ) ) ) ) ) ) ) )
```

CS1-4. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~2~1 validates/VBZ/validate
(SBAR~s~1~1 (S~s~2~2 (NPB~'~3~3 the/DT/the seller/NN/seller
'/POS/') (VP~s~2~1 s/VBZ/s (NPB~information~2~2
contact/NN/contact information/NN/information ./PUNC./.)))))
))

CS1-5. (TOP~asks~1~1 (S~asks~2~2 (NPB~System~1~1 System/NNP/System
) (VP~asks~3~1 asks/VBZ/ask (NPB~Supervisor~2~2 the/DT/the
Supervisor/NNP/Supervisor) (S~to~1~1 (VP~to~2~1 to/TO/to
(VP~validate~2~1 validate/VB/validate (NPB~seller~2~2 the/DT/the
seller/NN/seller ./PUNC./.)))))))

CS1-6. (TOP~permits~1~1 (S~permits~2~2 (NPB~Supervisor~1~1
Supervisor/NNP/Supervisor) (VP~permits~3~1 permits/VBZ/permit
(NPB~seller~2~2 the/DT/the seller/NN/seller) (S~to~1~1 (VP~to~2~1
to/TO/to (VP~operate~2~1 operate/VB/operate (PP~on~2~1 on/IN/on
(NPB~marketplace~2~2 the/DT/the marketplace/NN/marketplace
./PUNC./.))))))))

CS1-7. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~3~1 validates/VBZ/validate
(NPB~offer~3~3 the/DT/the whole/JJ/whole offer/NN/offer)
(PP~with~2~1 with/IN/with (NPB~Commission~3~3 the/DT/the
Trade/NNP/Trade Commission/NNP/Commission)))))

CS1-8. (TOP~lists~1~1 (S~lists~2~2 (NPB~System~1~1
System/NNP/System) (VP~lists~3~1 lists/VBZ/list (NPB~offer~2~2
the/DT/the offer/NN/offer) (PP~in~2~1 in/IN/in (NPB~offers~2~2
published/VBN/publish offers/NNS/offer ./PUNC./.)))))

CS1-9. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NPB~acknowledgment~3~3 an/DT/an
(ADJP~identified~2~2 uniquely/RB/uniquely identified/VBN/identify
) acknowledgment/NN/acknowledgment ./PUNC./.)))))

CS1-2a (TOP~performed~1~1 (S~performed~2~2 (NPB~Validation~1~1
Validation/NNP/Validation) (VP~performed~2~1
performed/VBD/perform (SBAR~by~2~1 by/IN/by (S~fails~2~2
(NPB~system~2~2 the/DT/the system/NN/system) (VP~fails~1~1
fails/VBZ/fail))))))

CS1-2a1 (TOP~aborted~1~1 (S~aborted~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~aborted~1~1 aborted/VBD/abort)))

CS1-2b (TOP~assessment~1~1 (NP~assessment~2~1 (NPB~assessment~2~2
Price/NNP/Price assessment/NN/assessment) (ADJP~available~1~1
available/JJ/available)))

CS1-2b1 (TOP~provides~1~1 (S~provides~2~2 (NPB~System~1~1
System/NNP/System) (VP~provides~3~1 provides/VBZ/provide
(NPB~seller~2~2 the/DT/the seller/NN/seller) (PP~with~2~1
with/IN/with (NPB~assessment~3~3 a/DT/a price/NN/price
assessment/NN/assessment ./PUNC./.)))))

CS1-7a (TOP~rejects~1~1 (S~rejects~2~2 (NPB~commission~2~2
Trade/NNP/Trade commission/NN/commission) (VP~rejects~2~1
rejects/VBZ/reject (NPB~offer~2~2 the/DT/the offer/NN/offer)))
)

```
CS1-7a1 (TOP~aborted~1~1 (S~aborted~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case ) (VP~aborted~1~1 aborted/VBD/abort ) ) )

CS2-1. (TOP~enters~1~1 (S~enters~2~2 (NPB~Buyer~1~1
Buyer/NNP/Buyer ) (VP~enters~2~1 enters/VBZ/enter
(NPB~criteria~3~3 basic/JJ/basic search/NN/search
criteria/NNS/criterion ./PUNC./. ) ) ) )

CS2-2. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System ) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NP~list~2~1 (NPB~list~2~2 the/DT/the
list/NN/list ) (PP~of~2~1 of/IN/of (NPB~matches~1~1
matches/VBZ/match ./PUNC./. ) ) ) ) ) ) )

CS2-3. (TOP~requests~1~1 (S~requests~2~2 (NPB~Buyer~1~1
Buyer/NNP/Buyer ) (VP~requests~2~1 requests/NNS/request
(NP~listing~2~1 (NPB~listing~3~3 the/DT/the complete/JJ/complete
listing/NN/listing ) (PP~of~2~1 of/IN/of (NPB~offer~3~3 a/DT/a
selected/VBN/select offer/NN/offer ./PUNC./. ) ) ) ) ) )

CS2-4. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System ) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NPB~information~3~3 the/DT/the
requested/VBN/request information/NN/information ./PUNC./. ) ) ) )
)

CS2-2a (TOP~matches~1~1 (S~matches~2~2 (NPB~No~1~1 No/DT/No )
(VP~matches~2~1 matches/VBZ/match (VP~found~1~1 found/VBN/find ) )
) )

CS2-2a1 (TOP~aborted~1~1 (S~aborted~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case ) (VP~aborted~1~1 aborted/VBD/abort ) ) )

CS2-2b (TOP~is~1~1 (S~is~2~2 (NP~amount~2~1 (NPB~amount~2~2
The/DT/The amount/NN/amount ) (PP~of~2~1 of/IN/of (NPB~matches~1~1
matches/VBZ/match ) ) ) (VP~is~2~1 is/VBZ/be (ADJP~high~2~2
too/RB/too high/JJ/high ) ) ) )

CS2-2b1 (TOP~narrows~1~1 (S~narrows~2~2 (NPB~Buyer~1~1
Buyer/NNP/Buyer ) (VP~narrows~2~1 narrows/VBZ/narrow
(NP~results~2~1 (NPB~results~3~3 the/DT/the search/NN/search
results/NNS/result ) (PP~with~2~1 with/IN/with (NPB~criteria~2~2
additional/JJ/additional criteria/NNS/criterion ) ) ) ) ) )

CS2-2b2 (TOP~Resume~1~1 (S~Resume~1~1 (VP~Resume~2~1
Resume/VB/Resume (PP~with~2~1 with/IN/with (NPB~step~2~1
step/NN/step 2/CD/2 ) ) ) ) )

CS3-1. (TOP~is~1~1 (S~is~2~2 (NPB~Clerk~1~1 Clerk/NNP/Clerk )
(VP~is~2~1 is/VBZ/be (VP~contacted~2~1 contacted/VBN/contact
(PP~by~2~1 by/IN/by (NP~buyer~2~1 (NPB~buyer~2~2 a/DT/a
buyer/NN/buyer ) (SBAR~who~2~1 (WHNP~who~1~1 who/WP/who )
(S~has~1~1 (VP~has~2~1 has/VBZ/have (VP~decided~2~1
decided/VBN/decide (S~to~1~1 (VP~to~2~1 to/TO/to (VP~accept~2~1
accept/VB/accept (NPB~offer~3~3 a/DT/a selected/VBN/select
offer/NN/offer ./PUNC./. ) ) ) ) ) ) ) ) ) ) ) ) )

CS3-2. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System ) (VP~validates~2~1 validates/VBZ/validate
(NPB~offer~2~2 the/DT/the offer/NN/offer ./PUNC./. ) ) ) )
```

CS3-3. (TOP~method~1~1 (NP~method~3~1 (NPB~method~8~8
System/NNP/System requests/NNS/request billing/NN/billing
and/CC/and shipping/NN/shipping information/NN/information
,/PUNC,/, payment/NN/payment method/NN/method) and/CC/and
(NPB~information~3~3 payment/NN/payment detail/NN/detail
information/NN/information ./PUNC./.)))

CS3-4. (TOP~enters~1~1 (S~enters~2~2 (NPB~Clerk~1~1
Clerk/NNP/Clerk) (VP~enters~3~1 enters/VBZ/enter
(NPB~information~2~2 billing/NN/billing information/NN/information
,/PUNC,/,) (S~select~1~1 (VP~select~3~1 (VP~select~2~1
select/VB/select (NPB~method~3~3 a/DT/a payment/NN/payment
method/NN/method)) and/CC/and (VP~provides~2~1
provides/VBZ/provide (NPB~details~3~3 the/DT/the
necessary/JJ/necessary details/NNS/detail ./PUNC./.)))))))

CS3-5. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~3~1 validates/VBZ/validate
(NPB~information~2~2 this/DT/this information/NN/information)
(PP~with~2~1 with/IN/with (NPB~Agency~4~4 a/DT/a Credit/NNP/Credit
Verification/NNP/Verification Agency/NNP/Agency ./PUNC./.))))
)

CS3-6. (TOP~performs~1~1 (S~performs~2~2 (NPB~System~1~1
System/NNP/System) (VP~performs~2~1 performs/VBZ/perform
(NPB~trade~2~2 the/DT/the trade/NN/trade ./PUNC./.))))

CS3-7. (TOP~informs~1~1 (S~informs~2~2 (NPB~System~1~1
System/NNP/System) (VP~informs~2~1 informs/VBZ/inform
(NP~seller~2~1 (NPB~seller~2~2 the/DT/the seller/NN/seller)
(SBAR~that~2~1 (WHNP~that~1~1 that/WDT/that) (S~has~2~2
(NPB~offer~2~2 the/DT/the offer/NN/offer) (VP~has~3~1 (VP~has~2~1
has/VBZ/have (VP~been~2~1 been/VBN/be (VP~accepted~1~1
accepted/VBN/accept))) and/CC/and (VP~provides~2~1
provides/VBZ/provide (NPB~information~3~3 the/DT/the
shipping/NN/shipping information/NN/information ./PUNC./.))))
)))))

CS3-8. (TOP~transfers~1~1 (S~transfers~2~2 (NPB~System~1~1
System/NNP/System) (VP~transfers~2~1 transfers/VBZ/transfer
(NP~account~4~4 (NP~payment~2~1 (NPB~payment~2~2 the/DT/the
payment/NN/payment) (PP~to~2~1 to/TO/to (NPB~seller~2~2
the/DT/the seller/NN/seller))) '/POS/' s/VBZ/s
account/NN/account ./PUNC./.))))

CS3-9. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~to~2~1 to/TO/to (NP~buyer~2~1 (NPB~buyer~2~2 the/DT/the
buyer/NN/buyer) (PP~with~2~1 with/IN/with (NPB~number~4~4
an/DT/an (ADJP~identified~2~2 uniquely/RB/uniquely
identified/VBN/identify) authorization/NN/authorization
number/NN/number ./PUNC./.)))))))

CS3-2a (TOP~is~1~1 (S~is~2~2 (NPB~Offer~1~1 Offer/NNP/Offer)
(VP~is~2~1 is/VBZ/be (ADJP~valid~2~2 not/RB/not valid/JJ/valid))
))

CS3-2a1 (TOP~is~1~1 (S~is~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~is~2~1 is/VBZ/be (ADJP~aborted~1~1
aborted/JJ/aborted))))

CS4-1. (TOP~locates~1~1 (S~locates~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~locates~2~1 locates/VBZ/locate
(NPB~offer~4~4 a/DT/a previously/RB/previously
submitted/VBN/submit offer/NN/offer ./PUNC./.))))

CS4-2. (TOP~requests~1~1 (NP~requests~2~1 (NPB~requests~2~2
Seller/NNP/Seller requests/NNS/request) (S~to~2~2 (NPB~system~2~2
the/DT/the system/NN/system) (VP~to~2~1 to/TO/to (VP~cancel~2~1
cancel/VB/cancel (NPB~offer~2~2 the/DT/the offer/NN/offer
./PUNC./.))))))

CS4-3. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NP~request~3~1 (NPB~request~2~2 a/DT/a
request/NN/request) (PP~for~2~1 for/IN/for (NPB~seller~2~2
the/DT/the seller/NN/seller)) (S~to~1~1 (VP~to~2~1 to/TO/to
(VP~prove~2~1 prove/VB/prove (NPB~identity~1~1
identity/NN/identity ./PUNC./.))))))))

CS4-4. (TOP~responds~1~1 (S~responds~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~responds~3~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NPB~number~3~3 the/DT/the
authorization/NN/authorization number/NN/number))
(VP~returned~2~1 returned/VBN/return (SBAR~when~2~1
(WHADVP~when~1~1 when/WRB/when) (S~was~2~2 (NPB~offer~2~2
the/DT/the offer/NN/offer) (VP~was~2~1 was/VBD/be
(VP~submitted.~1~1 submitted./VBN/submitted.)))))))

CS4-5. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~2~1 validates/VBZ/validate
(SBAR~s~1~1 (S~s~2~2 (NPB~'~5~5 the/DT/the request/NN/request
and/CC/and seller/NN/seller '/POS/') (VP~s~2~1 s/VBZ/s
(NPB~identity~1~1 identity/NN/identity ../PUNC./..)))))))

CS4-6. (TOP~removes~1~1 (S~removes~2~2 (NPB~System~1~1
System/NNP/System) (VP~removes~2~1 removes/VBZ/remove
(NPB~offer~2~2 the/DT/the offer/NN/offer ./PUNC./.))))

CS4-4a (TOP~provide~1~1 (S~provide~2~2 (NPB~cannot~2~2
Seller/NNP/Seller cannot/NN/cannot) (VP~provide~2~1
provide/VBP/provide (NPB~number~3~3 the/DT/the
authorization/NN/authorization number/NN/number))))

CS4-4a1 (TOP~is~1~1 (S~is~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~is~2~1 is/VBZ/be (ADJP~aborted~1~1
aborted/JJ/aborted))))

CS4-5a (TOP~is~1~1 (S~is~2~2 (NPB~number~2~2
Authorization/NNP/Authorization number/NN/number) (VP~is~2~1
is/VBZ/be (ADJP~valid~2~2 not/RB/not valid/JJ/valid))))

CS4-5a1 (TOP~Retry~1~1 (NP~Retry~2~1 (NPB~Retry~1~1 Retry/NN/Retry
) (PP~with~2~1 with/IN/with (NPB~step~2~1 step/NN/step 3/CD/3))
))

CS5-1. (TOP~locates~1~1 (S~locates~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~locates~2~1 locates/VBZ/locate
(NPB~offer~4~4 a/DT/a previously/RB/previously
submitted/VBN/submit offer/NN/offer ./PUNC./.))))

CS5-2. (TOP~requests~1~1 (NP~requests~2~1 (NPB~requests~2~2
Seller/NNP/Seller requests/NNS/request) (S~to~2~2 (NPB~system~2~2

the/DT/the system/NN/system) (VP~to~2~1 to/TO/to (VP~provide~2~1
provide/VB/provide (NP~status~2~1 (NPB~status~1~1 status/NN/status
) (PP~of~2~1 of/IN/of (NPB~offer~2~2 the/DT/the offer/NN/offer
./PUNC./.)))))))

CS5-3. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NP~request~3~1 (NPB~request~2~2 a/DT/a
request/NN/request) (PP~for~2~1 for/IN/for (NPB~seller~2~2
the/DT/the seller/NN/seller)) (S~to~1~1 (VP~to~2~1 to/TO/to
(VP~prove~2~1 prove/VB/prove (NPB~identity~1~1
identity/NN/identity ./PUNC./.))))))))

CS5-4. (TOP~responds~1~1 (S~responds~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~responds~3~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NPB~number~3~3 the/DT/the
authorization/NN/authorization number/NN/number))
(VP~returned~2~1 returned/VBN/return (SBAR~when~2~1
(WHADVP~when~1~1 when/WRB/when) (S~was~2~2 (NPB~offer~2~2
the/DT/the offer/NN/offer) (VP~was~2~1 was/VBD/be
(VP~submitted.~1~1 submitted./VBN/submitted.))))))))

CS5-5. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~2~1 validates/VBZ/validate
(SBAR~s~1~1 (S~s~2~2 (NPB~'~5~5 the/DT/the request/NN/request
and/CC/and seller/NN/seller '/POS/') (VP~s~2~1 s/VBZ/s
(NPB~identity~1~1 identity/NN/identity ../PUNC./..)))))))

CS5-6. (TOP~returns~1~1 (S~returns~2~2 (NPB~System~1~1
System/NNP/System) (VP~returns~2~1 returns/VBZ/return
(NP~status~2~1 (NPB~status~2~2 the/DT/the status/NN/status)
(PP~of~2~1 of/IN/of (NPB~offer~2~2 the/DT/the offer/NN/offer
./PUNC./.))))))

CS5-4a (TOP~provide~1~1 (S~provide~2~2 (NPB~cannot~2~2
Seller/NNP/Seller cannot/NN/cannot) (VP~provide~2~1
provide/VBP/provide (NPB~number~3~3 the/DT/the
authorization/NN/authorization number/NN/number))))

CS5-4a1 (TOP~is~1~1 (S~is~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~is~2~1 is/VBZ/be (ADJP~aborted~1~1
aborted/JJ/aborted))))

CS5-5a (TOP~is~1~1 (S~is~2~2 (NPB~number~2~2
Authorization/NNP/Authorization number/NN/number) (VP~is~2~1
is/VBZ/be (ADJP~valid~2~2 not/RB/not valid/JJ/valid))))

CS5-5a1 (TOP~Retry~1~1 (NP~Retry~2~1 (NPB~Retry~1~1 Retry/NN/Retry
) (PP~with~2~1 with/IN/with (NPB~step~2~1 step/NN/step 3/CD/3))
))

CS6-1. (TOP~locates~1~1 (S~locates~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~locates~2~1 locates/VBZ/locate
(NPB~offer~4~4 a/DT/a previously/RB/previously
submitted/VBN/submit offer/NN/offer ./PUNC./.))))

CS6-2. (TOP~requests~1~1 (NP~requests~2~1 (NPB~requests~2~2
Seller/NNP/Seller requests/NNS/request) (S~to~2~2 (NPB~system~2~2
the/DT/the system/NN/system) (VP~to~2~1 to/TO/to (VP~update~2~1
update/VB/update (NP~offer~2~1 (NPB~offer~2~2 the/DT/the
offer/NN/offer ,/PUNC,/,) (VP~providing~2~1 providing/VBG/provide
(NP~details~2~1 (NPB~details~2~2 new/JJ/new details/NNS/detail)

(PRN~-LRB-~3~1 -LRB-/-LRB-/-LRB- (NP~g~2~1 (NPB~g~2~2 e/NNP/e
./PUNC./. g/NNP/g ./PUNC./. ,/PUNC,/,) (NPB~price~1~1
price/NN/price)) -RRB-/-RRB-/-RRB- ./PUNC./.))))))))

CS6-3. (TOP~responds~1~1 (S~responds~2~2 (NPB~System~1~1
System/NNP/System) (VP~responds~2~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NP~request~3~1 (NPB~request~2~2 a/DT/a
request/NN/request) (PP~for~2~1 for/IN/for (NPB~seller~2~2
the/DT/the seller/NN/seller)) (S~to~1~1 (VP~to~2~1 to/TO/to
(VP~prove~2~1 prove/VB/prove (NPB~identity~1~1
identity/NN/identity ./PUNC./.))))))))

CS6-4. (TOP~responds~1~1 (S~responds~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~responds~3~1 responds/VBZ/respond
(PP~with~2~1 with/IN/with (NPB~number~3~3 the/DT/the
authorization/NN/authorization number/NN/number))
(VP~returned~2~1 returned/VBN/return (SBAR~when~2~1
(WHADVP~when~1~1 when/WRB/when) (S~was~2~2 (NPB~offer~2~2
the/DT/the offer/NN/offer) (VP~was~2~1 was/VBD/be
(VP~submitted.~1~1 submitted./VBN/submitted.)))))))

CS6-5. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~2~1 validates/VBZ/validate
(SBAR~s~1~1 (S~s~2~2 (NPB~'~5~5 the/DT/the request/NN/request
and/CC/and seller/NN/seller '/POS/') (VP~s~2~1 s/VBZ/s
(NPB~identity~1~1 identity/NN/identity ../PUNC./..)))))))

CS6-6. (TOP~updates~1~1 (S~updates~2~2 (NPB~System~1~1
System/NNP/System) (VP~updates~2~1 updates/VBZ/update
(NPB~offer~2~2 the/DT/the offer/NN/offer ./PUNC./.))))

CS6-4a (TOP~provide~1~1 (S~provide~2~2 (NPB~cannot~2~2
Seller/NNP/Seller cannot/NN/cannot) (VP~provide~2~1
provide/VBP/provide (NPB~number~3~3 the/DT/the
authorization/NN/authorization number/NN/number))))

CS6-4a1 (TOP~is~1~1 (S~is~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~is~2~1 is/VBZ/be (ADJP~aborted~1~1
aborted/JJ/aborted))))

CS6-5a (TOP~is~1~1 (S~is~2~2 (NPB~number~2~2
Authorization/NNP/Authorization number/NN/number) (VP~is~2~1
is/VBZ/be (ADJP~valid~2~2 not/RB/not valid/JJ/valid))))

CS6-5a1 (TOP~Retry~1~1 (NP~Retry~2~1 (NPB~Retry~1~1 Retry/NN/Retry
) (PP~with~2~1 with/IN/with (NPB~step~2~1 step/NN/step 3/CD/3))
))

CS7-1. (TOP~searches~1~1 (S~searches~2~2 (NPB~Buyer~1~1
Buyer/NNP/Buyer) (VP~searches~2~1 searches/VBZ/search (PP~for~2~1
for/IN/for (NPB~offer~2~2 an/DT/an offer/NN/offer)))))

CS7-2. (TOP~buys~1~1 (S~buys~3~3 (NPB~contacts~2~2 Buyer/NNP/Buyer
contacts/NNS/contact) (NP~clerk~2~1 (NPB~clerk~2~2 a/DT/a
clerk/NN/clerk) to/TO/to) (VP~buys~2~1 buys/VBZ/buy
(NPB~item~3~3 the/DT/the selected/VBN/select item/NN/item))))

CS7-1a (TOP~did~1~1 (S~did~2~2 (NPB~Buyer~2~2 The/DT/The
Buyer/NNP/Buyer) (VP~did~3~1 did/VBD/do not/RB/not (VP~find~2~1
find/VB/find (NPB~offer~3~3 any/DT/any matching/NN/matching
offer/NN/offer)))))

CS7-1a1 (TOP~aborted~1~1 (S~aborted~2~2 (NPB~case~2~2 Use/NNP/Use case/NN/case) (VP~aborted~1~1 aborted/VBD/abort)))

CS7-1b (TOP~decides~1~1 (S~decides~2~2 (NPB~Buyer~2~2 The/DT/The Buyer/NNP/Buyer) (VP~decides~2~1 decides/VBZ/decide (S~to~2~2 not/RB/not (VP~to~2~1 to/TO/to (VP~accept~2~1 accept/VB/accept (NPB~offer~2~2 the/DT/the offer/NN/offer ./PUNC./.)))))))

CS7-1b1 (TOP~ends~1~1 (S~ends~2~2 (NPB~case~2~2 Use/NNP/Use case/NN/case) (VP~ends~2~1 ends/VBZ/end (ADVP~here~1~1 here/RB/here ./PUNC./.))))

CS8-1. (TOP~searches~1~1 (S~searches~2~2 (NPB~Supervisor~1~1 Supervisor/NNP/Supervisor) (VP~searches~2~1 searches/VBZ/search (NP~database~2~1 (NPB~database~2~2 the/DT/the database/NN/database) (PP~of~2~1 of/IN/of (NP~offers~2~1 (NPB~offers~1~1 offers/NNS/offer) (PP~for~2~1 for/IN/for (NP~keywords~2~1 (NPB~keywords~2~2 sensitive/JJ/sensitive keywords/NNS/keyword) (PP~in~2~1 in/IN/in (NPB~description~2~2 item/NN/item description/NN/description)))))))))

CS8-2. (TOP~displays~1~1 (S~displays~2~2 (NPB~Supervisor~1~1 Supervisor/NNP/Supervisor) (VP~displays~2~1 displays/VBZ/display (NP~description~2~1 (NPB~description~2~2 the/DT/the description/NN/description) (PP~of~2~1 of/IN/of (NPB~item~2~2 the/DT/the item/NN/item ./PUNC./.))))))

CS8-3. (TOP~removes~1~1 (S~removes~2~2 (NPB~Supervisor~1~1 Supervisor/NNP/Supervisor) (VP~removes~2~1 removes/VBZ/remove (NP~item~2~1 (NPB~item~2~2 the/DT/the item/NN/item) (PP~from~2~1 from/IN/from (NP~database~2~1 (NPB~database~2~2 the/DT/the database/NN/database) (PP~of~2~1 of/IN/of (NPB~offers~3~3 currently/RB/currently visible/JJ/visible offers/NNS/offer ./PUNC./.))))))))

CS8-1a (TOP~did~1~1 (S~did~2~2 (NPB~Supervisor~1~1 Supervisor/NNP/Supervisor) (VP~did~3~1 did/VBD/do not/RB/not (VP~find~2~1 find/VB/find (NPB~match~2~2 any/DT/any match/NN/match)))))

CS8-1a1 (TOP~terminates~1~1 (S~terminates~2~2 (NPB~case~2~2 Use/NNP/Use case/NN/case) (VP~terminates~1~1 terminates/VBZ/terminate)))

CS8-2a (TOP~did~1~1 (S~did~2~2 (NPB~Supervisor~1~1 Supervisor/NNP/Supervisor) (VP~did~3~1 did/VBD/do not/RB/not (VP~find~2~1 find/VB/find (NPB~items~3~3 any/DT/any offending/JJ/offending items/NNS/item ./PUNC./.)))))

CS8-2a1 (TOP~terminates~1~1 (S~terminates~2~2 (NPB~case~2~2 Use/NNP/Use case/NN/case) (VP~terminates~1~1 terminates/VBZ/terminate)))

CS8-2b (TOP~requests~1~1 (S~requests~2~2 (NPB~Supervisor~1~1 Supervisor/NNP/Supervisor) (VP~requests~2~1 requests/NNS/request (NP~details~2~1 (NPB~details~1~1 details/NNS/detail) (PP~of~2~1 of/IN/of (NPB~item~2~2 another/DT/another item/NN/item))))))

CS8-2b1 (TOP~Repeat~1~1 (S~Repeat~2~1 (NPB~Repeat~1~1 Repeat/NN/Repeat) (NPB~step~2~1 step/NN/step 2/CD/2)))

CL1-1. (TOP~submits~1~1 (S~submits~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~submits~3~1 submits/VBZ/submit
(NPB~description~2~2 item/NN/item description/NN/description)
(PP~to~2~1 to/TO/to (NPB~clerk~2~2 the/DT/the clerk/NN/clerk
./PUNC./.)))))

CL1-2. (TOP~submits~1~1 (S~submits~2~2 (NPB~Clerk~1~1
Clerk/NNP/Clerk) (VP~submits~3~1 submits/VBZ/submit
(NPB~description~2~2 the/DT/the description/NN/description)
(PP~to~2~1 to/TO/to (NPB~system~2~2 the/DT/the system/NN/system
./PUNC./.)))))

CL1-3. (TOP~reports~1~1 (S~reports~2~2 (NPB~Clerk~1~1
Clerk/NNP/Clerk) (VP~reports~3~1 reports/VBZ/report
(NPB~system~2~2 the/DT/the system/NN/system) (NP~response~2~1
(NPB~response~1~1 response/NN/response) (PP~to~2~1 to/TO/to
(NPB~seller~2~2 the/DT/the seller/NN/seller ./PUNC./.))))))

CL1-4. (TOP~submits~1~1 (S~submits~2~2 (NPB~Seller~1~1
Seller/NNP/Seller) (VP~submits~3~1 submits/VBZ/submit
(NPB~information~6~6 the/DT/the price/NN/price ,/PUNC,/,
billing/NN/billing and/CC/and contact/NN/contact
information/NN/information) (PP~to~2~1 to/TO/to (NPB~clerk~2~2
the/DT/the clerk/NN/clerk ./PUNC./.)))))

CL1-5. (TOP~enters~1~1 (S~enters~2~2 (NPB~Clerk~1~1
Clerk/NNP/Clerk) (VP~enters~3~1 enters/VBZ/enter
(NPB~information~6~6 the/DT/the price/NN/price ,/PUNC,/,
billing/NN/billing and/CC/and contact/NN/contact
information/NN/information) (PP~to~2~1 to/TO/to (NPB~system~2~2
the/DT/the system/NN/system ./PUNC./.)))))

CL1-6. (TOP~reports~1~1 (S~reports~2~2 (NPB~Clerk~1~1
Clerk/NNP/Clerk) (VP~reports~3~1 reports/VBZ/report
(NPB~system~2~2 the/DT/the system/NN/system) (NP~response~2~1
(NPB~response~1~1 response/NN/response) (PP~to~2~1 to/TO/to
(NPB~seller~2~2 the/DT/the seller/NN/seller ./PUNC./.))))))

CL1-2a (TOP~performed~1~1 (S~performed~2~2 (NPB~Validation~1~1
Validation/NNP/Validation) (VP~performed~2~1
performed/VBD/perform (SBAR~by~2~1 by/IN/by (S~fails~2~2
(NPB~system~2~2 the/DT/the system/NN/system) (VP~fails~1~1
fails/VBZ/fail))))))

CL1-2a1 (TOP~aborted~1~1 (S~aborted~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~aborted~1~1 aborted/VBD/abort)))

CL2-1. (TOP~submits~1~1 (S~submits~2~2 (NPB~Buyer~1~1
Buyer/NNP/Buyer) (VP~submits~3~1 submits/VBZ/submit (PP~to~2~1
to/TO/to (NPB~clerk~2~2 the/DT/the clerk/NN/clerk))
(NP~reference~2~1 (NPB~reference~2~2 a/DT/a reference/NN/reference
) (PP~to~2~1 to/TO/to (NPB~offer~3~3 a/DT/a selected/VBN/select
offer/NN/offer ./PUNC./.))))))

CL2-2. (TOP~submits~1~1 (S~submits~2~2 (NPB~Clerk~1~1
Clerk/NNP/Clerk) (VP~submits~2~1 submits/VBZ/submit
(NP~reference~2~1 (NPB~reference~2~2 the/DT/the
reference/NN/reference) (PP~to~2~1 to/TO/to (NPB~system~2~2
the/DT/the system/NN/system ./PUNC./.))))))

CL2-3. (TOP~reports~1~1 (S~reports~2~2 (NPB~Clerk~1~1
Clerk/NNP/Clerk) (VP~reports~3~1 reports/VBZ/report

(NPB~system~2~2 the/DT/the system/NN/system) (NP~response~5~1
(NP~response~2~1 (NPB~response~1~1 response/NN/response)
(PP~to~2~1 to/TO/to (NPB~seller~2~2 the/DT/the seller/NN/seller)
)) and/CC/and (NPB~billing~2~2 requests/NNS/request
billing/NN/billing) and/CC/and (NP~method~4~2 (S~shipping~1~1
(VP~shipping~2~1 shipping/VBG/ship (NPB~information~1~1
information/NN/information ,/PUNC,/,))) (NPB~method~2~2
payment/NN/payment method/NN/method) and/CC/and (NPB~details~2~2
payment/NN/payment details/NNS/detail ./PUNC./.))))))

CL2-4. (TOP~submits~1~1 (S~submits~2~2 (NPB~Buyer~1~1
Buyer/NNP/Buyer) (VP~submits~3~1 submits/VBZ/submit (PP~to~2~1
to/TO/to (NPB~clerk~2~2 the/DT/the clerk/NN/clerk))
(NPB~details~11~11 the/DT/the requested/VBN/request
billing/NN/billing and/CC/and shipping/NN/shipping
information/NN/information ,/PUNC,/, payment/NN/payment
method/NN/method and/CC/and payment/NN/payment details/NNS/detail
./PUNC./.))))

CL2-5. (TOP~enters~1~1 (S~enters~2~2 (NPB~Clerk~1~1
Clerk/NNP/Clerk) (VP~enters~2~1 enters/VBZ/enter (NP~method~3~1
(NPB~method~7~7 the/DT/the billing/NN/billing and/CC/and
shipping/NN/shipping information/NN/information ,/PUNC,/,
payment/NN/payment method/NN/method) and/CC/and (NPB~details~2~2
payment/NN/payment details/NNS/detail ./PUNC./.)))))

CL2-6. (TOP~reports~1~1 (S~reports~2~2 (NPB~Clerk~1~1
Clerk/NNP/Clerk) (VP~reports~4~1 reports/VBZ/report
(NPB~system~2~2 the/DT/the system/NN/system) (NP~response~2~1
(NPB~response~1~1 response/NN/response) (PRN~-LRB-~3~1
-LRB-/-LRB-/-LRB- (PP~with~2~1 with/IN/with
(NPB~acknowledgment~3~3 the/DT/the unique/JJ/unique
acknowledgment/NN/acknowledgment)) -RRB-/-RRB-/-RRB-))
(PP~to~2~1 to/TO/to (NPB~buyer~2~2 the/DT/the buyer/NN/buyer
./PUNC./.)))))

CL2-3a (TOP~failed~1~1 (S~failed~2~2 (NPB~System~1~1
System/NNP/System) (VP~failed~2~1 failed/VBD/fail (S~to~1~1
(VP~to~2~1 to/TO/to (VP~validate~2~1 validate/VB/validate
(NPB~offer~2~2 the/DT/the offer/NN/offer))))))

CL2-3a1 (TOP~aborted~1~1 (S~aborted~2~2 (NPB~case~2~2 Use/NNP/Use
case/NN/case) (VP~aborted~1~1 aborted/VBD/abort)))

SU1-1. (TOP~asks~1~1 (S~asks~2~2 (NPB~system~2~2
Computer/NN/Computer system/NN/system) (VP~asks~3~1 asks/VBZ/ask
(NPB~supervisor~2~2 the/DT/the supervisor/NN/supervisor)
(S~to~1~1 (VP~to~2~1 to/TO/to (VP~decide~2~1 decide/VB/decide
(PP~on~2~1 on/IN/on (S~permitting~1~1 (VP~permitting~3~1
permitting/VBG/permit (NPB~seller~2~2 a/DT/a seller/NN/seller)
(S~to~1~1 (VP~to~2~1 to/TO/to (VP~operate~2~1 operate/VB/operate
(PP~on~2~1 on/IN/on (NPB~marketplace~2~2 the/DT/the
marketplace/NN/marketplace ./PUNC./.)))))))))))))

SU1-2. (TOP~validates~1~1 (S~validates~2~2 (NPB~System~1~1
System/NNP/System) (VP~validates~3~1 (VP~validates~2~1
validates/VBZ/validate (NPB~seller~2~2 the/DT/the seller/NN/seller
)) and/CC/and (VP~signals~3~1 signals/VBZ/signal (NPB~system~2~2
the/DT/the system/NN/system) (S~to~1~1 (VP~to~2~1 to/TO/to
(VP~permit~2~1 permit/VB/permit (S~to~2~2 (NPB~seller~2~2
the/DT/the seller/NN/seller) (VP~to~2~1 to/TO/to (VP~operate~1~1
operate/VB/operate ./PUNC./.)))))))))

SU2-1. (TOP~requests~1~1 (NP~requests~2~1 (NPB~requests~2~2 Supervisor/NNP/Supervisor requests/NNS/request) (S~to~2~2 (NPB~system~3~3 the/DT/the computer/NN/computer system/NN/system) (VP~to~2~1 to/TO/to (VP~search~2~1 search/VB/search (NP~database~2~1 (NPB~database~2~2 the/DT/the database/NN/database) (PP~of~2~1 of/IN/of (NP~offers~2~1 (NPB~offers~1~1 offers/NNS/offer) (PP~for~2~1 for/IN/for (NP~keywords~2~1 (NPB~keywords~2~2 sensitive/JJ/sensitive keywords/NNS/keyword) (PP~in~2~1 in/IN/in (NPB~description~2~2 item/NN/item description/NN/description)))))))))))

SU2-2. (TOP~found~1~1 (S~found~3~3 (NP~requests~2~1 (NPB~requests~2~2 Supervisor/NNP/Supervisor requests/NNS/request) (PP~from~2~1 from/IN/from (NPB~system~3~3 the/DT/the computer/NN/computer system/NN/system))) (NP~descriptions~2~1 (NPB~descriptions~2~2 detailed/JJ/detailed descriptions/NNS/description) (PP~of~2~1 of/IN/of (NPB~item~2~2 an/DT/an item/NN/item))) (VP~found~1~1 found/VBD/find ./PUNC./.)))

SU2-3. (TOP~requests~1~1 (NP~requests~2~1 (NPB~requests~2~2 Supervisor/NNP/Supervisor requests/NNS/request) (S~to~2~2 (NPB~system~3~3 the/DT/the computer/NN/computer system/NN/system) (VP~to~2~1 to/TO/to (VP~remove~3~1 remove/VB/remove (NPB~item~2~2 the/DT/the item/NN/item) (PP~from~2~1 from/IN/from (NP~database~2~1 (NPB~database~2~2 the/DT/the database/NN/database) (PP~of~2~1 of/IN/of (NPB~offers~3~3 currently/RB/currently visible/JJ/visible offers/NNS/offer ./PUNC./.)))))))))

SU2-1a (TOP~found~1~1 (S~found~2~2 (NPB~item~3~3 No/DT/No matching/VBG/match item/NN/item) (VP~found~1~1 found/VBD/find)))

SU2-1a1 (TOP~terminates~1~1 (S~terminates~2~2 (NPB~case~2~2 Use/NNP/Use case/NN/case) (VP~terminates~1~1 terminates/VBZ/terminate)))

SU2-2a (TOP~is~1~1 (S~is~2~2 (NPB~item~3~2 The/DT/The item/NN/item does/VBZ/do) (VP~is~3~1 is/VBZ/be not/RB/not (NPB~item~3~3 an/DT/an offending/JJ/offending item/NN/item ./PUNC./.))))

SU2-2a1 (TOP~terminates~1~1 (S~terminates~2~2 (NPB~case~2~2 Use/NNP/Use case/NN/case) (VP~terminates~1~1 terminates/VBZ/terminate)))

SU2-2b (TOP~requests~1~1 (S~requests~2~2 (NPB~Supervisor~1~1 Supervisor/NNP/Supervisor) (VP~requests~2~1 requests/NNS/request (NP~details~2~1 (NPB~details~1~1 details/NNS/detail) (PP~of~2~1 of/IN/of (NPB~item~2~2 another/DT/another item/NN/item))))))

SU2-2b1 (TOP~Repeat~1~1 (S~Repeat~2~1 (NPB~Repeat~1~1 Repeat/NN/Repeat) (NPB~step~2~1 step/NN/step 2/CD/2)))

Appendix D: Procasor Tool Output: Event Tokens and Pro-cases

In this appendix, we provide the output obtained with our tool. In Sect. D.1, we list the event tokens obtained for each use case step; the Pro-cases assembled from these event tokens are shown in Sect. D.2. Next, Sect. D.3 contains the list of all events to be processed by each of the involved entity; finally, in Sect. D.4 we show the frame Pro-case assembled from the Pro-cases obtained with the Procasor tool.

D.1. Event Tokens

This section lists the event tokens obtained with the Procasor tool for each parse tree of a use case step (App. C). In the prototype implementation, this output is stored in the file `output/tokens.txt`. Note that as the tool internally sorts the steps lexicographically by their label, the order in the output differs from the order in the input file (listed in Appendix C).

```
CL1-1     ?SL.submitItem
CL1-2     #submitDescription
CL1-2a1   %ABORT
CL1-3     !SL.reportSystem
CL1-4     ?SL.submitPrice
CL1-5     #enterPrice
CL1-6     !SL.reportSystem
CL2-1     ?BU.submitClerk
CL2-2     #submitReference
CL2-3     !SL.reportSystem
CL2-3a1   %ABORT
CL2-4     ?BU.submitClerk
CL2-5     #enterPaymentMethod
CL2-6     !BU.reportSystem
CS1-1     ?CL.submitInformation
CS1-2     #validateDescription
CS1-2a1   %ABORT
CS1-2b1   !SL.providePriceAssessment
CS1-3     ?CL.adjustPrice
CS1-4     #validateSeller
CS1-5     !SU.validateSeller
CS1-6     ?SU.permitSeller
CS1-7     !TC.validateOffer
CS1-7a1   %ABORT
CS1-8     #listOffer
CS1-9     #respondAcknowledgment
CS2-1     ?BU.enterSearchCriterion
CS2-2     #respondList
CS2-2a1   %ABORT
CS2-2b1   ?BU.narrowSearchResult
CS2-2b2   %GOTO 2
CS2-3     ?BU.listing
CS2-4     #respondInformation
CS3-1     ?CL.contactBuyer
CS3-2     #validateOffer
CS3-2a1   %ABORT
CS3-3     #paymentDetailInformation
CS3-4     ?CL.enterBillingInformation
CS3-5     !CRA.validateInformation
CS3-6     #performTrade
CS3-7     !SL.informOffer
```

```
CS3-8     #transferPayment
CS3-9     !BU.respondAuthorizationNumber
CS4-1     ?SL.locateOffer
CS4-2     ?SL.cancelOffer
CS4-3     !SL.respondRequest
CS4-4     ?SL.respondAuthorizationNumber
CS4-4a1   %ABORT
CS4-5     #validateRequestSeller
CS4-5a1   %GOTO 3
CS4-6     #removeOffer
CS5-1     ?SL.locateOffer
CS5-2     ?SL.provideStatus
CS5-3     !SL.respondRequest
CS5-4     ?SL.respondAuthorizationNumber
CS5-4a1   %ABORT
CS5-5     #validateRequestSeller
CS5-5a1   %GOTO 3
CS5-6     #returnStatus
CS6-1     ?SL.locateOffer
CS6-2     ?SL.updateOffer
CS6-3     !SL.respondRequest
CS6-4     ?SL.respondAuthorizationNumber
CS6-4a1   %ABORT
CS6-5     #validateRequestSeller
CS6-5a1   %GOTO 3
CS6-6     #updateOffer
CS7-1     ?BU.searchOffer
CS7-1a1   %ABORT
CS7-1b1   %ABORT
CS7-2     ?BU.buyItem
CS8-1     ?SU.searchDatabase
CS8-1a1   %ABORT
CS8-2     ?SU.displayDescription
CS8-2a1   %ABORT
CS8-2b1   %GOTO 2
CS8-3     ?SU.removeItem
M1-1      ?SL.submitItem
M1-2      #validateDescription
M1-2a1    %ABORT
M1-2b1    !SL.providePriceAssessment
M1-3      ?SL.adjustPrice
M1-4      #validateSeller
M1-5      #verifySeller
M1-5a1    %ABORT
M1-6      !TC.validateOffer
M1-6a1    %ABORT
M1-7      #listOffer
M1-8      #respondAuthorizationNumber
M2-1      ?BU.enterSearchCriterion
M2-2      #respondList
M2-2a1    %ABORT
M2-2b1    ?BU.narrowSearchResult
M2-2b2    %GOTO 2
M2-3      ?BU.listing
M2-4      #respondInformation
M3-1      ?BU.acceptOffer
M3-2      #validateOffer
M3-2a1    %ABORT
M3-3      ?BU.enterBillingInformation
M3-4      !CRA.validateBuyer
M3-5      #performSale
M3-6      !SL.informOffer
```

```
M3-7      #transferPayment
M3-8      !BU.respondAuthorizationNumber
M4-1      ?SL.locateOffer
M4-2      ?SL.cancelOffer
M4-3      !SL.respondRequest
M4-4      ?SL.respondAuthorizationNumber
M4-4a1    %ABORT
M4-5      #validateRequestSeller
M4-5a1    %GOTO 3
M4-6      #removeOffer
M5-1      ?SL.locateOffer
M5-2      ?SL.provideStatus
M5-3      !SL.respondRequest
M5-4      ?SL.respondAuthorizationNumber
M5-4a1    %ABORT
M5-5      #validateRequestSeller
M5-5a1    %GOTO 3
M5-6      #returnStatus
M6-1      ?SL.locateOffer
M6-2      ?SL.updateOffer
M6-3      !SL.respondRequest
M6-4      ?SL.respondAuthorizationNumber
M6-4a1    %ABORT
M6-5      #validateRequestSeller
M6-5a1    %GOTO 3
M6-6      #updateOffer
M7-1      ?BU.searchOffer
M7-1a1    %ABORT
M7-1b1    %ABORT
M7-2      ?BU.buyItem
SU1-1     ?CS.decideSupervisor
SU1-2     #validateSeller
SU2-1     !CS.searchDatabase
SU2-1a1   %ABORT
SU2-2     !CS.findDescription
SU2-2a1   %ABORT
SU2-2b1   %GOTO 2
SU2-3     !CS.removeItem
```

D.2. Pro-cases

This section lists the Pro-cases obtained with the Procasor tool in the Marketplace case study. In the prototype implementation, this output is stored in the file `output/procases.txt`.

```
CL1  ?SL.submitItem;#submitDescription;(#cond2a+
     !SL.reportSystem;?SL.submitPrice;#enterPrice;
     !SL.reportSystem)
CL2  ?BU.submitClerk;#submitReference;!SL.reportSystem;(
     #cond3a+?BU.submitClerk;#enterPaymentMethod;
     !BU.reportSystem)
CS1  ?CL.submitInformation;#validateDescription;(#cond2a+
     (NULL+#cond2b;!SL.providePriceAssessment);
     ?CL.adjustPrice;#validateSeller;!SU.validateSeller;
     ?SU.permitSeller;!TC.validateOffer;
     (#cond7a+#listOffer;#respondAcknowledgment))
CS2  ?BU.enterSearchCriterion;#respondList;(#cond2b;
     ?BU.narrowSearchResult;#respondList)*;
     (#cond2a+?BU.listing;#respondInformation)
CS3  ?CL.contactBuyer;#validateOffer;(#cond2a+
     #paymentDetailInformation;?CL.enterBillingInformation;
     !CRA.validateInformation;#performTrade;!SL.informOffer;
     #transferPayment;!BU.respondAuthorizationNumber)
CS4  ?SL.locateOffer;?SL.cancelOffer;(!SL.respondRequest;
     ?SL.respondAuthorizationNumber;#validateRequestSeller;
     #cond5a)*;!SL.respondRequest; ?SL.respondAuthorizationNumber;
     (#cond4a+#validateRequestSeller;#removeOffer)
CS5  ?SL.locateOffer;?SL.provideStatus;(!SL.respondRequest;
     ?SL.respondAuthorizationNumber;#validateRequestSeller;
     #cond5a)*;!SL.respondRequest; ?SL.respondAuthorizationNumber;
     (#cond4a+#validateRequestSeller;#returnStatus)
CS6  ?SL.locateOffer;?SL.updateOffer;(!SL.respondRequest;
     ?SL.respondAuthorizationNumber;#validateRequestSeller;
     #cond5a)*;!SL.respondRequest; ?SL.respondAuthorizationNumber;
     (#cond4a+#validateRequestSeller;#updateOffer)
CS7  ?BU.searchOffer;(#cond1a+#cond1b+?BU.buyItem)
CS8  ?SU.searchDatabase;(#cond1a+(?SU.displayDescription;
     #cond2b)*;?SU.displayDescription; (#cond2a+?SU.removeItem))
M1   ?SL.submitItem;#validateDescription;(#cond2a+(NULL+
     #cond2b;!SL.providePriceAssessment);?SL.adjustPrice;
     #validateSeller;#verifySeller;(#cond5a+!TC.validateOffer;(#co
     nd6a+#listOffer;#respondAuthorizationNumber)))
M2   ?BU.enterSearchCriterion;#respondList;(#cond2b;
     ?BU.narrowSearchResult;#respondList)*;
     (#cond2a+?BU.listing;#respondInformation)
M3   ?BU.acceptOffer;#validateOffer;(#cond2a+
     ?BU.enterBillingInformation;!CRA.validateBuyer;
     #performSale;!SL.informOffer;#transferPayment;
     !BU.respondAuthorizationNumber)
M4   ?SL.locateOffer;?SL.cancelOffer;(!SL.respondRequest;
     ?SL.respondAuthorizationNumber;#validateRequestSeller;
     #cond5a)*;!SL.respondRequest;
     ?SL.respondAuthorizationNumber;(#cond4a+
     #validateRequestSeller;#removeOffer)
M5   ?SL.locateOffer;?SL.provideStatus;(!SL.respondRequest;
     ?SL.respondAuthorizationNumber;#validateRequestSeller;
     #cond5a)*;!SL.respondRequest; ?SL.respondAuthorizationNumber;
     (#cond4a+#validateRequestSeller;#returnStatus)
```

```
M6   ?SL.locateOffer;?SL.updateOffer;(!SL.respondRequest;
     ?SL.respondAuthorizationNumber;#validateRequestSeller;
     #cond5a)*;!SL.respondRequest; ?SL.respondAuthorizationNumber;
     (#cond4a+#validateRequestSeller;#updateOffer)
M7   ?BU.searchOffer;(#cond1a+#cond1b+?BU.buyItem)
SU1  ?CS.decideSupervisor;#validateSeller
SU2  !CS.searchDatabase;(#cond1a+(!CS.findDescription;
     #cond2b)*;!CS.findDescription;(#cond2a+!CS.removeItem))
```

D.3. Lists of Operations on Interfaces

To demonstrate the potential of how the transformation implemented in the Procasor tool could be employed in a CASE tool to aid with creating the initial design, we construct the list of all operations to be received, sent and internally processed by each entity. This section contains these lists obtained in the Marketplace case study; in the prototype implementation, these lists are stored in the file `output/operations.txt`.

```
Entity BU
Provided operations:
  reportSystem  CL
  respondAuthorizationNumber  CS,M

Required operations:
  submitClerk  CL
  enterSearchCriterion CS,M
  narrowSearchResult  CS,M
  listing  CS,M
  searchOffer  CS,M
  buyItem  CS,M
  acceptOffer  M
  enterBillingInformation  M

Internal operations:

Entity CL
Provided operations:
  submitItem  SL
  submitPrice  SL
  submitClerk  BU

Required operations:
  reportSystem  BU,SL
  submitInformation  CS
  adjustPrice  CS
  contactBuyer  CS
  enterBillingInformation  CS

Internal operations:
  submitDescription
  enterPrice
  submitReference
  enterPaymentMethod

Entity CRA
Provided operations:
  validateInformation  CS
  validateBuyer M

Required operations:

Internal operations:

Entity CS
Provided operations:
  submitInformation  CL
  adjustPrice  CL
  permitSeller  SU
  enterSearchCriterion BU
  narrowSearchResult  BU
```

```
   listing   BU
   contactBuyer  CL
   enterBillingInformation   CL
   locateOffer   SL
   cancelOffer   SL
   respondAuthorizationNumber   SL
   provideStatus SL
   updateOffer   SL
   searchOffer   BU
   buyItem   BU
   searchDatabase   SU
   displayDescription   SU
   removeItem  SU
   findDescription  SU

Required operations:
   providePriceAssessment  SL
   validateSeller   SU
   validateOffer TC
   validateInformation  CRA
   informOffer   SL
   respondAuthorizationNumber   BU
   respondRequest   SL
   decideSupervisor   SU

Internal operations:
   validateDescription
   validateSeller
   listOffer
   respondAcknowledgment
   respondList
   respondInformation
   validateOffer
   paymentDetailInformation
   performTrade
   transferPayment
   validateRequestSeller
   removeOffer
   returnStatus
   updateOffer

Entity M
Provided operations:
   submitItem  SL
   adjustPrice   SL
   enterSearchCriterion BU
   narrowSearchResult   BU
   listing   BU
   acceptOffer   BU
   enterBillingInformation   BU
   locateOffer   SL
   cancelOffer   SL
   respondAuthorizationNumber   SL
   provideStatus SL
   updateOffer   SL
   searchOffer   BU
   buyItem   BU

Required operations:
   providePriceAssessment  SL
   validateOffer TC
   validateBuyer CRA
```

```
   informOffer   SL
   respondAuthorizationNumber   BU
   respondRequest   SL

Internal operations:
   validateDescription
   validateSeller
   verifySeller
   listOffer
   respondAuthorizationNumber
   respondList
   respondInformation
   validateOffer
   performSale
   transferPayment
   validateRequestSeller
   removeOffer
   returnStatus
   updateOffer

Entity MKT
Provided operations:

Required operations:

Internal operations:

Entity SL
Provided operations:
   reportSystem  CL
   providePriceAssessment  CS,M
   informOffer   CS,M
   respondRequest   CS,M

Required operations:
   submitItem  CL,M
   submitPrice   CL
   locateOffer   CS,M
   cancelOffer   CS,M
   respondAuthorizationNumber   CS,M
   provideStatus CS,M
   updateOffer   CS,M
   adjustPrice   M

Internal operations:

Entity SU
Provided operations:
   validateSeller   CS
   decideSupervisor   CS

Required operations:
   permitSeller  CS
   searchDatabase   CS
   displayDescription   CS
   removeItem  CS
   findDescription  CS

Internal operations:
   validateSeller
```

```
Entity TC
Provided operations:
   validateOffer CS,M

Required operations:

Internal operations:
```

D.4. Frame Pro-case from Automatically Created Use Cases

In Appendix A, we demonstrated the frame Pro-case for the entity Marketplace Information System (M) assembled from the manually created Pro-cases, according to the Pro-case expression derived in Chapter 4. The Procasor tool described in Chapter 5 constructs the Pro-cases for textual use cases in an automated way; however, the behavior assembly (deriving the Pro-case expression) still remains a manual process. We reuse the Pro-case expression already derived for these use cases in Chapter 4; by substituting the automatically created Pro-cases (App. D.2) into the expression; we obtain the following frame Pro-case. Note that the current prototype implementation does not consider included use cases; as use-case M7 includes use cases M2 and M3, we substituted their Pro-cases manually for the two actions constructed from steps 1 and 2 of use case M7 (renaming the conditional events to avoid name clashes).

Thus, for the frame Pro-case expression

$$PE^{M}_{R} = (PC^{M}_{1} + PC^{M}_{4} + PC^{M}_{5} + PC^{M}_{6})\ \mathbf{*}\ \|\ (PC^{M}_{7})\ \mathbf{*}$$

we obtain the frame Pro-case:

```
(
?SL.submitItem;#validateDescription;(#cond2a+(NULL+
#cond2b;!SL.providePriceAssessment);?SL.adjustPrice;
#validateSeller;#verifySeller;(#cond5a+!TC.validateOffer;
(#cond6a+#listOffer;#respondAuthorizationNumber)))
+
?SL.locateOffer;?SL.cancelOffer;(!SL.respondRequest;
?SL.respondAuthorizationNumber;#validateRequestSeller;
#cond5a)*;!SL.respondRequest;
?SL.respondAuthorizationNumber;(#cond4a+
#validateRequestSeller;#removeOffer)
+
?SL.locateOffer;?SL.provideStatus;(!SL.respondRequest;
?SL.respondAuthorizationNumber;#validateRequestSeller;
#cond5a)*;!SL.respondRequest;?SL.respondAuthorizationNumber;
(#cond4a+#validateRequestSeller;#returnStatus)
+
?SL.locateOffer;?SL.updateOffer;(!SL.respondRequest;
?SL.respondAuthorizationNumber;#validateRequestSeller;
#cond5a)*;!SL.respondRequest;?SL.respondAuthorizationNumber;
(#cond4a+#validateRequestSeller;#updateOffer)
) *
||
(
?BU.enterSearchCriterion;#respondList;(#condM22b;
?BU.narrowSearchResult;#respondList)*;
(#condM22a+?BU.listing;#respondInformation)
;( #condM71a+#condM71b+?BU.acceptOffer;#validateOffer;(#cond
M32a+?BU.enterBillingInformation;!CRA.validateBuyer;
#performSale;!SL.informOffer;#transferPayment;
!BU.respondAuthorizationNumber) )
) *
```