

Automatically Deriving a UML Analysis Model from a Use Case Model

by

Tao Yue, M.A.Sc.

A thesis submitted to the Faculty of Graduate Studies and Research
In partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Electrical and Computer Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering (OCIECE)
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
September 2010

Copyright © 2010 by Tao Yue



Library and Archives
Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*
ISBN: 978-0-494-70564-3
Our file *Notre référence*
ISBN: 978-0-494-70564-3

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.


Canada

Abstract

The transition from requirements expressed in Natural Language (NL) to a structured, precise specification is an important challenge in practice. It is particularly so for object-oriented methods, defined in the context of the OMG's Model Driven Architecture (MDA). However, its automation has received little attention, mostly because requirements are in practice expressed in NL and are much less structured than other kinds of development artifacts. Such an automated transformation would enable at least the generation of an initial, likely incomplete, analysis model and enable automated traceability from requirements to code, through various intermediate models. In this thesis, we propose a method and a tool, building on existing work, to automatically generate analysis models from requirements. Such models include class, sequence and activity diagrams to describe the structure and behavior of a system. It also includes automatically establishing traceability links between the requirements and the automatically generated analysis model. Requirements are assumed to be modeled with our use case modeling approach (RUCM), which we showed to have enough expressive power, to be easy to apply, to help improve the quality of manually derived analysis models.

Seven (six) case studies were performed to compare class (sequence) diagrams generated by our tool to the ones created by experts, Masters students, and trained, 4th year undergraduate students. Our results show that our method performs well when compared to reference diagrams. Further, statistical test results show that our tool significantly outperforms 4th year engineering students, thus demonstrating the value of automation. Performance analysis results show that the execution time of the tool remains within a range of a few minutes, thus suggesting the approach is scalable. Five case studies were performed to assess our approach's capability to generate activity diagrams. The results show that high quality activity diagrams can be generated and the analysis also shows that our approach outperforms existing academic approaches and commercial tools. We also conducted two industrial case studies, yielding results that demonstrate that RUCM

is applicable in two different industrial domains and that a Toucan-generated analysis models are largely correct and complete from the perspective of domain experts.

Dedication

To the loving memory of my father who instilled in me the confidence that I have the capability to do anything that I have in my mind.

To my mother for her truly unconditional and forever love.

Acknowledgements

I would like to show my deep and sincere gratitude to my supervisor, Professor Lionel C. Briand for guiding me through the process of transitioning from an engineer into a researcher. Throughout my Masters and PhD studies, he taught me how to be a good researcher and I am also really proud of being one of his students. I must also express my warm and sincere thanks to my supervisor, Professor Yvan Labiche for all the valuable advice and discussion on almost every single detail of my work and paper preparation. I wish to thank all my colleagues at the SQUALL laboratory for providing a nice working environment. Thank you all for those valuable memories. I am indebted to my entire family and all my friends, who are always there for me no matter what happens.

THANK YOU ALL!

TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	1
CHAPTER 2	SYSTEMATIC REVIEW	6
2.1	Systematic review scope	7
2.2	Research questions	7
2.3	Search strategy	8
2.3.1	Identification of query string	9
2.3.2	Electronic and manual search	10
2.3.3	Inclusion/exclusion criteria	11
2.3.4	Statistics from included primary studies.....	12
2.4	Conceptual framework	13
2.4.1	Static model	13
2.4.2	Taxonomies	15
2.4.3	Process model	20
2.5	Evaluation Criteria	22
2.5.1	Evaluation criterion for requirements	23
2.5.2	Evaluation criterion for analysis models	24
2.5.3	Evaluation criteria for transformation	24
2.5.4	Discussion.....	26
2.6	Synthesis and Evaluation	27
2.6.1	Requirements configurations.....	29
2.6.2	Analysis models.....	30
2.6.3	Transformation - automation	31
2.6.4	Transformation - efficiency.....	33
2.6.5	Transformation – others	34
2.7	Open issues and suggestions	44
2.7.1	Approach configuration	44
2.7.2	Intermediate model	49
2.7.3	Transformations.....	50
2.8	Conclusion	53
2.9	Other related work	55
CHAPTER 3	RUCM	57
3.1	Related work	57
3.1.1	Use case template.....	57
3.1.2	Restriction rules	60
3.2	Use case template	61
3.3	Restriction rules	64
3.4	Example	66
CHAPTER 4	UCMETA	69
4.1	Architecture	70

4.2 Packages	71
4.2.1 UCSTemplate.....	72
4.2.2 SentencePatterns.....	74
4.2.3 SentenceSemantics	76
4.2.4 SentenceStructure	77
CHAPTER 5 FORMALIZEUCM	80
5.1 Transformation rules	80
5.1.1 UCMeta, UML::UseCases and UCSTemplate.....	80
5.1.2 SentenceStructure	81
5.1.3 SentencePatterns and SentenceSemantics.....	82
5.2 Algorithm.....	84
5.2.1 ParseUCM and ParseUCS	84
5.2.2 ParseSimpleSentence.....	85
5.3 Traceability	86
CHAPTER 6 GENERATEUML	88
6.1 Rule Set A: Structure	88
6.2 Algorithm.....	89
6.3 Traceability	89
CHAPTER 7 GENERATE CLASS DIAGRAMS	91
7.1 Transformation rules	91
7.2 Algorithm.....	97
CHAPTER 8 GENERATE SEQUENCE DIAGRAMS	100
8.1 Transformation rules	100
8.2 Algorithms	105
CHAPTER 9 GENERATE ACTIVITY DIAGRAMS	107
9.1 Overview	108
9.2 Transformation Rules.....	110
9.3 Algorithm.....	116
9.4 Traceability	117
CHAPTER 10 AUTOMATION (ATOUCAN)	119
10.1 FormalizeUCM subsystem.....	120
10.2 GenerateUML subsystem	121
CHAPTER 11 EVALUATION FRAMEWORK.....	123
CHAPTER 12 EVALUATION OF RUCM	125
12.1 Related work	125

12.2 Experiment Planning	126
12.2.1 Experiment definition	127
12.2.2 Context selection and subjects	128
12.2.3 Hypotheses formulation	129
12.2.4 Experiment design	130
12.2.5 Instrumentation	133
12.2.6 Evaluation measurement and data collection	140
12.3 Experiment Results and analysis	149
12.3.1 Usage of restriction rules	151
12.3.2 Quality of analysis models	156
12.4 Threats to Validity	176
12.5 Conclusion	178
CHAPTER 13 EVALUATION OF CLASS DIAGRAMS	180
13.1 Design of the study	181
13.2 Evaluation results	182
13.3 Discussion	184
13.4 Performance analysis	184
CHAPTER 14 EVALUATION OF SEQUENCE DIAGRAMS	186
14.1 Design of study	187
14.2 Evaluation results and analysis	187
CHAPTER 15 EVALUATION OF ACTIVITY DIAGRAMS	191
15.1 Validation procedure and summary of results	191
15.2 Comparison with three commercial tools	193
15.3 Related Work and Comparison	195
CHAPTER 16 INDUSTRIAL CASE STUDIES	200
CHAPTER 17 SUMMARY	204
17.1 Conclusion	204
17.2 Future research directions	206
REFERENCES	207

LIST OF FIGURES

Figure 1 Static model (class Traceability Link appears four times for layout purposes) .	15
Figure 2 Taxonomy of requirements.....	16
Figure 3 Taxonomy of restriction rules	18
Figure 4 Taxonomy of analysis models.....	19
Figure 5 Taxonomy of requirements preprocessing approaches	20
Figure 6 Taxonomy of transformation approaches.....	20
Figure 7 A generic transformation process.....	22
Figure 8 Mapping between the conceptual framework and evaluation criteria and summaries	23
Figure 9 Architecture of UCMeta.....	71
Figure 10 Package UCMeta of UCMeta	71
Figure 11 Example of package UCMeta	72
Figure 12 Package Template of UCMeta	73
Figure 13 Package ComplexSentence of UCMeta.....	74
Figure 14 Package SpecialSentence of UCMeta.....	74
Figure 15 Example of package SentenceStructure.....	76
Figure 16 Package SentenceStructure of UCMeta.....	77
Figure 17 Package Phrase of UCMeta	78
Figure 18 Metaclass NounPhrase and its functional constituents	79
Figure 19 An example of parse tree and dependencies.....	82
Figure 20 Formal specification/identification of SVDO.....	83
Figure 21 Formal specification/identification of Initiation	83
Figure 22 ParseUCM operation	85
Figure 23 ParseSimpleSentence operation.....	86
Figure 24 Traceability model.....	90
Figure 25 Class diagram automatically generated by aToucan for the ATM system (Withdraw Fund Control class).....	91
Figure 26 Class diagram automatically generated by aToucan for the ATM system (entity classes).....	92
Figure 27 Transformation algorithm for generating class diagrams.....	98

Figure 28 Transformation algorithm for generating class diagrams - invoke_rule_B1....	98
Figure 29 Transformation algorithm for generating class diagrams - invoke_rule_B2....	99
Figure 30 Transformation algorithm for generating sequence diagrams.....	106
Figure 31 Running example – activity diagram generation.....	109
Figure 32 Transformation rule D1.2.3 in the Kermeta language [55]	115
Figure 33 Transformation algorithm for generating activity diagrams	117
Figure 34 Architecture of the FormalizeUCM subsystem.....	121
Figure 35 Architecture of the GenerateUML subsystem.....	122
Figure 36 Evaluation framework	124
Figure 37 Error rates of restriction rules (when $\geq 5\%$)	152
Figure 38 Understandability of the restriction rules (when $<90\%$)	153
Figure 39 High applicability of the restriction rules (when $< 90\%$)	154
Figure 40 High restrictiveness of the restriction rules (when $> 20\%$)	155
Figure 41 Distributions of Class Completeness - Labs 1&2.....	158
Figure 42 Distributions of CD Completeness - Labs 1&2.....	158
Figure 43 Distributions of Class Correctness - Labs 1&2	159
Figure 44 Distributions of Redundancy - Labs 1&2.....	159
Figure 45 Distributions of Class Completeness - Lab 2.1	160
Figure 46 Distributions of CD Completeness - Lab 2.1	160
Figure 47 Distributions of Class Correctness - Lab 2.1.....	161
Figure 48 Distributions of Redundancy - Lab 2.1	161
Figure 49 Distributions of Class Completeness - Lab 2.3	161
Figure 50 Distributions of CD Completeness - Lab 2.3	162
Figure 51 Distributions of Class Correctness - Lab 2.3.....	162
Figure 52 Distributions of Redundancy - Lab 2.3	162
Figure 53 Distributions of Message Completeness - Lab 2.2.....	168
Figure 54 Distributions of SD Completeness - Lab 2.2.....	168
Figure 55 Distributions of Message Correctness - Lab 2.2	168
Figure 56 Distributions of SD Correctness - Lab 2.2	169
Figure 57 Distributions of BCE Consistency - Lab 2.2	169
Figure 58 Distributions of Message Completeness - Lab 2.4.....	170

Figure 59 Distributions of SD Completeness - Lab 2.4.....	170
Figure 60 Distributions of Message Correctness - Lab 2.4	170
Figure 61 Distributions of SD Correctness - Lab 2.4	171
Figure 62 Distributions of BCE Consistency - Lab 2.4.....	171
Figure 63 Taxonomy of sentence pattern SV.....	252
Figure 64 Taxonomy of sentence pattern SVDO.....	254
Figure 65 Taxonomy of sentence pattern SVIODO	255
Figure 66 Taxonomy of sentence pattern SVDOC.....	258
Figure 67 Taxonomy of sentence pattern SLVSubjectComplt	260
Figure 68 Formal specification/identification of Function	264
Figure 69 Formal specification/identification of Initiation	264
Figure 70 Formal specification/identification of Validation	265
Figure 71 Formal specification/identification of InternalTransaction.....	265
Figure 72 Formal specification/identification of Response2PrimaryActor.....	266
Figure 73 Formal specification/identification of Response2SecondaryActor ...	266
Figure 74 ATM System - control classes.....	270
Figure 75 ATM System - entity classes.....	271
Figure 76 ATM System - Query Account Control class.....	271
Figure 77 ATM System - Transfer Fund Control class	272
Figure 78 ATM System - Withdraw Fund Control class.....	272
Figure 79 ATM System - Validate PIN Control class	273
Figure 80 ARENA system – part 1	273
Figure 81 ARENA system – part 2	274
Figure 82 Elevator system – entity classes	275
Figure 83 Elevator system – control classes.....	275
Figure 84 Elevator system – Request Elevator Control.....	276
Figure 85 Video Store system – control classes	276
Figure 86 Video Store system – entity classes.....	277
Figure 87 Video Store system – boundary classes.....	277
Figure 88 Video Store system – Check database Control.....	278
Figure 89 Video Store system – Rent video Control.....	278

Figure 90 Video Store system –Return video Control.....	278
Figure 91 Video Store system – Return video Control.....	279
Figure 92 Video Store system – Video overdue Control.....	279
Figure 93 Cab Dispatching system – control classes.....	280
Figure 94 Cab Dispatching system – entity classes.....	280
Figure 95 Cab Dispatching system – entity classes.....	281
Figure 96 Payroll system – control classes.....	281
Figure 97 Payroll system – entity classes.....	282
Figure 98 Payroll system – boundary classes.....	283
Figure 99 CPD system – entity classes.....	284
Figure 100 ATM System – Withdraw Funds (sequence diagram in the textbook)	285
Figure 101 Sequence diagram generated for the basic flow of use case Withdraw Fund (in part).....	286
Figure 102 Sequence diagram generated for the global alternative flow of use case Withdraw Funds.....	287
Figure 103 Sequence diagram generated for the bounded alternative flow of use case Withdraw Funds.....	287
Figure 104 Sequence diagram generated for the specific alternative flow of use case Withdraw Funds.....	288
Figure 105 Overview Activity Diagram – aToucan	290
Figure 106 Detailed Activity Diagram – aToucan.....	291
Figure 107 Re-written UCS - Visual Paradigm (partial)	292
Figure 108 Activity Diagram - Visual Paradigm (partial).....	293
Figure 109 Activity Diagram - Ravenflow (for better layout, the activity diagram is exported to RSA and visualized by it).....	294
Figure 110 Re-written UCS - Ravenflow	294
Figure 111 Re-written UCS - CaseComplete (partial).....	295
Figure 112. Activity Diagram - CaseComplete (partial)	296
Figure 113 Least-square means for ANOVA interaction analysis between Method and System for CD - Lab 2.1	329
Figure 114 Least-square means for ANOVA interaction analysis between Method and System for CD - Lab 2.3	330

Figure 115 Least-square means for ANOVA interaction analysis between Method and Order for CD - Lab 2.1 and Lab 2.3	331
Figure 116 Least-square means for ANOVA interaction analysis between Method and System for SD - Lab 2	332
Figure 117 Least-square means for ANOVA interaction analysis between Method and System for SD - Lab 2.4	333
Figure 118 Least-square means for ANOVA interaction analysis between Method and Order for SD - Lab 2.2 and Lab 2.4.....	334

LIST OF TABLES

Table 1 Summary of electronic search results	12
Table 2 Evaluation summary of the approaches proposed in the primary studies.....	28
Table 3 Restriction rules on requirements documentation	41
Table 4 Summary of transformation rules (part 1)	42
Table 5 Summary of transformation rules (part 2)	43
Table 6 Use case template.....	63
Table 7 Restriction rules (R1-R7).....	64
Table 8 Restriction rules (R8-R16).....	65
Table 9 Restriction rules (R17-R26).....	66
Table 10 Use case Withdraw Funds.....	68
Table 11 Summary of rule Set B1.....	95
Table 12 Summary of rule Set B2.....	96
Table 13 Summary of rule set C	104
Table 14 Summary of rule set D	112
Table 15 Goal 1.....	127
Table 16 Goal 2.....	127
Table 17 Hypotheses – Experiment 2	130
Table 18 Participant groups and tasks – Experiment 1	131
Table 19 Participant groups and tasks – Experiment 2.....	133
Table 20 Comprehension questionnaire of Task 1	135
Table 21 Classification and examples of comprehension questions.....	139
Table 22 Question distributions of the comprehension questionnaires for Task 2.....	139
Table 23 Measures used to derive Error Rate.....	141
Table 24 Error Rate measurements (R1–R16).....	142
Table 25 Error Rate measurements (R17–R26).....	142
Table 26 Measures used to derive CD	144
Table 27 Quality measures for a class	145
Table 28 Quality measures for a class diagram	145
Table 29 Measures used to derive SD.....	148
Table 30 Quality measures for a sequence diagram	149

Table 31 Collected and analyzed data points – Experiment 2	151
Table 32 Spearman nonparametric correlation analysis – correlation table	156
Table 33 Two tailed <i>t</i> -test and Wilcoxon test– CD.....	164
Table 34 Summary of <i>t</i> -test results of CD	165
Table 35 Summary of ANOVA tests - CD.....	166
Table 36 One tailed <i>t</i> -test and Wilcoxon test– SD.....	172
Table 37 Summary of <i>t</i> -test results of SD.....	174
Table 38 Summary of ANOVA tests - SD.....	174
Table 39 Descriptive statistics of CS – lab 2.2 and lab 2.4	175
Table 40 One way <i>t</i> -test – CS – lab 2.2 and lab 2.4.....	175
Table 41 Descriptive statistics of QC – Experiment 1, Experiment 2: lab 2.2, and lab 2.3	175
Table 42 <i>t</i> -test – QC – Experiment 1, Experiment 2: lab 2.2, and lab 2.3	175
Table 43 Characteristics of each case study system – Class Diagram.....	181
Table 44 Evaluation results of all measures – aToucan (class diagram)	183
Table 45 One sample <i>t</i> -test –4 th students and aToucan (class diagram)	184
Table 46 Performance analysis results (class diagram)	185
Table 47 Characteristics of each case study system - sequence diagram	187
Table 48 Evaluation results of aToucan against experts – sequence diagram.....	189
Table 49 Evaluation results of 4 th students against aToucan – sequence diagram	190
Table 50 Completeness of data flow information.....	193
Table 51 Evaluation summary (part I).....	197
Table 52 Evaluation summary (part II).....	197
Table 53 Characteristics of the generated analysis models of each industrial case study system	202
Table 54 Responses from domain experts on the evaluation questions.....	203
Table 55 Restrictions (part 1)	215
Table 56 Restrictions (part 2)	216
Table 57 Restrictions (part 3)	217
Table 58 Restrictions (part 4)	218
Table 59 Data flow informatin on detailed activity diagram.....	291
Table 60 Experiment 1 – Comprehension Questionnaire – Part 1 (R1-R16)	298

Table 61 Experiment 1 – Comprehension Questionnaire – Part 2 (R17-R26)	299
Table 62 Error rates of the restriction rules	322
Table 63 Understandability of the restriction rules.....	322
Table 64 Applicability of the restriction rules (frequencies).....	322
Table 65 Restrictiveness of the restriction rules (frequencies).....	323
Table 66 Descriptive statistics of measure CD – Lab 2.1 and 2.3	324
Table 67 Descriptive statistics of measure SD – Lab 2.2 and 2.4	325
Table 68 ANOVA – Interaction between Method (UCM_R vs. UCM_UR) and System (CPD vs. VS) for CD – Lab 2.1 and Lab 2.3	326
Table 69 ANOVA – Interaction between Method (UCM_R vs. UCM_UR) and Order (2.1 vs. 2.3) on CD – Lab 2.1 and Lab 2.3	326
Table 70 ANOVA – Interaction between Method (UCM_R vs. UCM_UR) and System (CPD vs. VS) for SD – Experiment 2	327
Table 71 ANOVA – Interaction between Method (UCM_R vs. UCM_UR) and Order (2.2 vs. 2.4) on SD – Lab 2.2 and Lab 2.4	328

LIST OF ACRONYMS

aToucan	Automated Transformation Of Use Case model into ANalysis model
EMF	Eclipse Modeling Framework
FormalizeUCM	Formalize Use Case Specifications
GenerateUML	Generate UML analysis models
MDA	Model Driven Architecture
NL	Natural Language
NLP	Natural Language Processing
NP	Noun Phrase
OCL	Object Constraint Language
PIM	Platform Independent Model
POS	Part-Of-Speech
PSM	Platform Specific Model
RUCM	Restricted Use Case Modeling
UCMod	Use Case Model
UCMeta	Use Case Metamodel
UCS	Use Case Specification
UML	Unified Modeling Language

CHAPTER 1 INTRODUCTION

Transforming requirements expressed in Natural Language (NL) into a structured, precise specification is an important problem both to support analysis and to provide support for subsequent phases of development. Two main research schools have been working on this problem area. The first is to propose techniques and tools to transform NL-based requirements into propositional or first-order logic properties to enable requirements completeness checking (e.g., [103]), temporal properties to verify them against a system design or implementation (e.g., [66]), or formal context tables for reconciling requirements originating from different stakeholders (e.g., [86]). The second school, in the context of model-driven development, focuses instead on approaches to transform textual requirements into models, to facilitate transition to subsequent phases of development. Our work is part of this latter category.

Model transformation is the fundamental principle underpinning Model Driven Architecture (MDA) [72]. To build a software system, a series of transformations is performed: from the requirements to the Platform Independent Model (PIM) (the analysis model), from the PIM to the Platform Specific Model (PSM) (the design model), and from the PSM to source code. Though much work (e.g., [45, 72]) has focused on the last two transformations, the first one has received limited attention, partly because it is not clearly identified as being part of the MDA lifecycle [57] that is often depicted as starting from an analysis model and ending with a deployed executable code. The most likely reason for such limited amount of research is that requirements are mostly textual, and are therefore much less structured than other kinds of software development artifacts. The fully automated manipulation of requirements is therefore highly difficult. In other words, textual requirements are not easily amenable to complete automated transformations. However, requirements engineering is an obvious necessity of software development. Therefore automated transformation from requirements to analysis models, even if partial, would provide significant help in making MDA cost effective. Additionally, automated transformation would enable automated traceability from requirements to code (through the PIM and the PSM). In this context, traceability is the ability to link

requirements to corresponding analysis and design model elements, and then code, test cases, and other software artifacts.

Our overall objective is to devise a method and tool to automatically generate an (initial) software analysis model, in the Unified Modeling Language (UML) [73], based on textual, functional requirements documented as a Use Case Model (UCMod). A second objective is to establish traceability links between requirements and analysis model elements. An analysis model describes the software system (or simply system) under development at a high level of abstraction, describing its logical components and behavior. In a typical software development lifecycle (e.g., [45]), UML analysis models typically include class and sequence diagrams that describe structure and component interactions, respectively. Our research aims to generate such a UML analysis model automatically, though it is expected that such a model will be an initial version to be manually refined by analysts. Notice that we assume a specific, use case-driven software development methodology, specifically a methodology similar to what is promoted by prominent textbook authors like Dutoit and Gomaa in their textbooks [15] and [39], respectively. We acknowledge that there exist other types of requirements representations (e.g., use case map [6]) than use case models and other types of requirements engineering methodologies (e.g., KAOS [60]) than use case driven approaches. These are however out of the scope of this thesis.

Our approach builds on the results of a systematic literature review (Chapter 2) on transformations of textual requirements into analysis models. We identified a number of variation factors in existing approaches leading to different drawbacks and strengths. These factors include (1) whether the widely popular use cases or other concepts were used to structure requirements, (2) what use case template was adopted, (3) what NL restrictions, if any, were applied when specifying use cases, (4) whether domain specific information (e.g., glossary, domain model) was used, (5) if the approach relied on one or more intermediary models between requirements and the analysis model. After pondering carefully all factors, based on existing studies, we selected the following strategy to strike what we thought was the best balance between practicality, level of automation, and requirements engineering effort: a) We do not depend on any domain specific

information; b) We rely on UCMods and a specific template to structure textual Use Case Specifications (UCSs); c) The restrictions we propose for NL and UCSs build on our systematic literature review and appear to be easy to use based on experimental results (Chapter 12); d) We use UML as the notation for analysis models, consistent with the MDA standard [57], which requires the source and target of a transformation be represented as UML models; e) Only one intermediate model is used; f) We rely on transformation rules that are well structured and precisely specified, so that they are easy to extend and modify; g) Our method establishes traceability links during transformations.

The basis of our method is a use case modeling approach called RUCM (Chapter 3), which is based on a use case template and a set of appropriate restriction rules for textual UCSs. Using template and restriction rules is a common feature of NL analysis approaches for reducing imprecision and incompleteness in UCSs. The main factor to consider is that the restricted NL should be expressive and convenient enough for use by developers. We have conducted a controlled experiment (Chapter 12) to evaluate RUCM and the results demonstrate that RUCM has enough expressive power, is easy to apply, helps achieve better understandability of use cases and improves the quality of manually derived analysis models.

Our method has three sequential steps. During the first step, requirements are manually defined by following RUCM (Chapter 3). The second step is to analyze textual UCSs (output of step one) to identify Part-Of-Speech (POS) and grammatical relation dependencies of sentences in UCSs, and then record that information into an instance of the intermediate metamodel of our method (UCMeta), as described in Chapter 4 and Chapter 5. The third step is to transform the instance of UCMeta into an analysis model that is an instance of the UML 2.0 metamodel: Chapter 6 describes the overall transformation, Chapter 7, Chapter 8, and Chapter 9 present the generation of class, sequence and activity diagrams, respectively. The last two steps are automatically performed by our tool, called aToucan (Automated Transformation Of Use Case model into ANalysis model) (Chapter 10). aToucan is rule-based and, thanks to a modular

design, facilitates modifications to the set of transformation rules to accommodate different contexts (e.g., use different parsers).

We also propose an evaluation framework (Chapter 11) to evaluate our use case modeling approach (RUCM) from the aspects of its applicability and its impact on the quality of derived class and sequence diagrams from RUCM models (Chapter 12), and aToucan automatically generated class diagrams (Chapter 13), sequence diagrams (Chapter 14), and activity diagrams (Chapter 15).

Seven case studies were performed to compare class diagrams generated by aToucan to class diagrams created by textbook authors (considered as experts), experts from our modeling courses, and trained 4th year undergraduate students (Chapter 13). Results indicate that our approach performs well with respect to consistency and completeness when compared to the reference class diagram, e.g., 92% class diagram consistency and 73% class completeness. aToucan also significantly outperforms trained 4th year undergraduate students in many aspects (e.g., class consistency and completeness). Four case study systems have been used to evaluate sequence diagrams generated by aToucan against the ones devised by experts (Chapter 14). Results indicate that, for all the six use cases we used, the sequence diagrams generated by aToucan are highly consistent with the ones devised by experts, and are also very complete, e.g., 91% and 97% message consistency and completeness, respectively. We also compared the sequence diagrams manually created by 30 fourth-year undergraduate students with the ones automatically generated by aToucan for two of those case study systems. Results show that, for the five use cases we used, diagrams generated by aToucan are far more complete (+46%) than the ones manually generated by students. Therefore, overall, aToucan helps generate useful, initial analysis models. In addition, two industrial case studies in the domains of communication systems and integrated control systems were conducted to demonstrate the applicability of RUCM in at least two industrial domains (Chapter 16). The main question was to investigate whether aToucan generated analysis models that were meaningful from the perspective of domain experts.

The systematic review and some other related work not included in the systematic review is described in Chapter 2. RUCM is presented in Chapter 3, followed by the use case metamodel (UCMeta) (Chapter 4). The transformation (FormalizeUCM) from a textual UCMMod to a formalized UCMMod (instance of UCMeta) is presented in Chapter 5. In Chapter 6 to Chapter 9, we present how our approach transforms a formalized UCMMod into a UML analysis model including class, sequence and activity diagrams. The implementation of our approach (aToucan) is given in Chapter 10. In Chapter 11 to Chapter 16, we show how we evaluate our approach. Last, we conclude the thesis in Chapter 17.

CHAPTER 2 **SYSTEMATIC REVIEW**

We conducted a systematic literature review on transformations of textual requirements into analysis models. The review identified 20 primary studies (16 approaches) based on a carefully designed paper selection procedure in scientific journals and conferences from 1996 to 2008 and Software Engineering textbooks. The method proposed here is based on the results of this review, integrates existing work, and addresses open issues that are discussed in the remainder of this section.

A systematic review follows a well-defined method to identify, analyze, synthesize, evaluate, and compare all available literature works relevant to a specific research topic [56]. A systematic review is an important piece of work since it summarizes existing techniques concerning a research interest, it identifies further research directions, and it provides a framework to position new research activities [56]. Several discrete activities (i.e., guidelines) are recommended for all systematic reviews in software engineering [56], among which are four recommended essential steps that we adopted: First, we specify our systematic review scope (Section 2.1), followed by defining research questions that the systematic review is expecting to answer (Section 2.2); Second, we develop a search strategy (Section 2.3), which includes a paper selection procedure, resources to be searched, and inclusion and exclusion criteria; Third, we use the search strategy to identify the relevant research works; Last, we analyze, synthesize, and compare the related works to answer the research questions.

The main objective of this systematic review is to develop a conceptual framework (Section 2.4) to synthesize and compare the related works proposing approaches of transformation between requirements and analysis models (Section 2.6) with a set of evaluation criteria (Section 2.5). Based on the synthesis and comparison results, we summarize and analyze the reviewed works, identify open issues, and provide suggestions for future research (Section 2.7). We conclude the section in Section 2.8.

2.1 Systematic review scope

In this section, we refine the scope of the systematic review by defining, more precisely, the relevant, fundamental concepts.

A requirement is defined in [109] as “a statement that identifies a necessary attribute, capability, characteristic, or quality of a system in order for it to have value and utility to a user”. Requirements should be easy to understand since they are usually written as a means for communication between different stakeholders (e.g., users, developers). There are many different ways to document requirements. One common way is to use textual descriptions only. Other ways to document requirements include use cases and customized document templates. For some systems (e.g., safety critical systems), requirements may even be documented as formal specifications. In our systematic review, we limit our scope to requirements documented using pure textual descriptions, use cases, or customized document templates. We exclude formal methods (mathematical descriptions) from our systematic review since they are less often used to formalize requirements in practice and present very different problems than textual requirements (i.e., mapping between formal languages).

An analysis model is a description of what a system is required to do functionally, and aims to be less ambiguous and more correct and consistent than textual requirements [14]. In a typical object-oriented software development process, the analysis model is usually derived from requirements. It is typically represented as a UML model containing various diagrams and possibly constraints. Our systematic review is, however, not limited to UML models. Other representations are also taken into account, as they often share similar object-oriented concepts. Other well-known notations include, for example, Message Sequence Charts (MSC) or Entity Relationship Models (ERM).

2.2 Research questions

In order to examine the evidence of transforming requirements into analysis models using different transformation approaches, this systematic review aims to answer the following research questions: (1) What are the different approaches used for transforming

requirements into analysis models? (2) What are the current limitations of these approaches? (3) What are the open issues to be further investigated?

The first research question is further divided into the following sub-questions: (i) What are the different requirement representations (e.g., use cases, pure textual specifications, and customer-specified templates) required by these approaches? Is it difficult for users to document such requirements? Is there any tool support? (ii) What kinds of analysis models (e.g., UML diagrams and Message Sequence Charts) can be generated by these approaches? Does a generated analysis model contain both the structural and behavioral aspects of a system? (iii) Are there any intermediate models used during transformation from requirements to analysis models? How do they affect the efficiency of the transformation? (iv) Are these approaches automated, automatable, semi-automated, or manual? Are there algorithms presented in the approaches? (v) What steps are taken by each approach to transform user requirements into analysis models? (vi) Do approaches include traceability management support? (vii) Have any case studies been performed to evaluate the approaches? If yes, what results have been obtained? What other evaluation methods (besides case studies) have been applied to evaluate these approaches?

2.3 Search strategy

In this section, we present the search strategy that we applied to select papers to be reviewed. We initiated our search by identifying a query string being used to perform electronic searches, based on our research questions (Section 2.3.1). Identifying our query string was an iterative process. We started with a query string and used it in five electronic databases, making sure (validation of the query) that the result contained papers we already knew about. As a tremendous number of papers resulted from the search, we tried to reduce the scope of the search by refining the query string. This is the reason why, for instance, we do not have keyword “use case” alone in the query string; instead the keyword is used in combination with other terms (e.g., use case analysis) to have a more focused search. Then, we searched five electronic databases using this query string (Section 2.3.2). In addition, as a complement to the electronic search, we performed manual search in specific journals and conference proceedings, and also

manually checked Software Engineering textbooks (Section 2.3.2). We then scanned all the sources resulting from this two-stage search to select the works to be included in the review. During this step we applied inclusion/exclusion criteria (Section 2.3.3) to select primary studies. For each paper, we read the paper's title and abstract to see whether it was relevant to our research topic. If the title and abstract of the paper could not help us make a decision, we further checked the paper's full text. In order to augment our collection of primary studies, we scanned the reference lists of all the identified primary studies to identify additional papers. Furthermore, we also went through publication lists of primary studies' authors to make sure that the most recent publications on the same or similar topics were included. The statistic data of included primary studies is presented in Section 2.3.4.

2.3.1 Identification of query string

Based on the research questions (Section 2.2), we identified three groups of search terms: population terms, intervention terms, and outcome terms. The population terms are the keywords that represent the domain of transforming requirements into analysis models, such as requirements analysis, requirements refinement, use cases realization, and domain modeling. The intervention terms are the keywords that represent the techniques applied in the population to achieve an objective. In our case, the techniques for transforming requirements into analysis models could be very different; therefore we decided to use general terms like transformation, generation, and linguistic analysis. The outcome terms represent different types of analysis models, which could be generated.

To form the query string, we used a disjunction of the keywords of each term group, and then used the conjunction of the three groups of terms. The following three groups of terms were used to form the query string:

Population terms: requirements analysis; requirements engineering; requirements refinement; requirements formalization; use cases analysis; use cases formalization; use cases realization; object oriented analysis; object-oriented analysis; object identification; domain modeling.

Intervention terms: automated transformation; automatic transformation; transformation; transform; transforming; translation; translate; translating; derive; deriving; generation; generate; generating; linguistic analysis; linguistic analyze; natural language processing.

Outcome terms: analysis model(s); object model(s); static model(s); dynamic model(s); UML model(s); class diagram(s); sequence diagrams(s); interaction diagram(s); activity diagrams(s); state machine(s); statechart(s); class model(s); interaction model(s); object oriented model(s); object-oriented model(s); object(s); class(es); message sequence chart(s).

2.3.2 Electronic and manual search

We performed electronic search within five electronic databases: IEEE Xplore, ACM Digital Library, Compendex, Inspec, and SpringerLink; using the query string we described earlier. Each time, we modified the query string to fit the format requirements of the electronic database before applying it. We also manually searched all published papers from 1996 to 2008 in nine potentially relevant, peer-reviewed journals: IEEE Transactions on Software Engineering, Automated Software Engineering, Requirements Engineering Journal, Journal of Natural Language Engineering, ACM Transactions on Software Engineering and Methodology, Journal of Systems and Software, Software and Systems Modeling, Information and Software Technology, and Data & Knowledge Engineering. We also manually searched all published papers from 1996 to 2008 in five potentially related conference proceedings: ACM/IEEE International Conference on Software Engineering, IEEE International Conference on Software Maintenance, IEEE/ACM International Conference on Automated Software Engineering, IEEE International Conference on Model Driven Engineering Languages and Systems and the former UML workshops, and IEEE International Requirements Engineering Conference. We also manually searched Software Engineering textbooks (e.g., [14, 61]) that describe transformations from requirements to analysis models.

2.3.3 Inclusion/exclusion criteria

Approaches for transforming requirements into analysis models vary a great deal as different requirement representations are adopted as inputs, and/or different analysis models are generated. Therefore, it is absolutely necessary that we define thorough inclusion/exclusion criteria to select the primary studies that can answer our research questions (Section 2.2) and also conform to our research scope (Section 2.1). We used the following inclusion/exclusion criteria:

- Papers irrelevant to the transformation of requirements are excluded. For example, the papers discussing information retrieval techniques that are used to recover traceability links (one of the capabilities of transformation approaches) between software artifacts are excluded.
- Papers proposing transformation approaches between software artifacts that are out of our review scope are excluded. For example, transformations between requirements given as formal specifications and design models or code are excluded (e.g., [99]).
- When encountering more than one paper describing the same or similar approaches, which were published in different venues, we only included the most recent one or the one with the most complete description of the approach.
- When a single approach is presented in more than one paper describing different parts of the approach, we included all these papers, but still considered them as a single approach.
- Papers with insufficient technical information regarding their approaches were excluded. For example, the papers that do not provide a detailed description on requirements representations, intermediate models (if any), and transformation techniques, are considered incomplete and are excluded (e.g., [20]).
- Software Engineering textbooks (e.g., [14, 60-62, 84, 96]) share similarities with respect to the requirements to analysis model transformation. They all describe an

intuitive set of heuristics to guide designers identify objects, attributes, and associations from a requirement specification, mapping parts of speech (e.g., nouns, verbs and adjectives) to model elements (e.g., objects, operations, and attributes). Some of these books refer to Abbott’s heuristics [3] (e.g., [14, 61, 84, 96]), others provide similar heuristics to Abbott’s (e.g., [62]), and a third group extends Abbott’s heuristics (e.g., [14, 60]). These textbooks suggest that these heuristics be applied manually (no transformation approach is described) and no requirements pre-processing technique is applied. According to our taxonomy of approaches (Section 2.4.2), all those approaches fall into the same category. Since Abbott’s heuristics is the primary study that is mostly used or referenced, we therefore only include Abbott’s heuristics [3] as one of the primary studies. We only mention individual textbooks when we discuss their transformation rules (heuristics) that do not refer to Abbott’s or extend Abbott’s.

2.3.4 Statistics from included primary studies

The electronic search results are summarized in Table 1. A total of 451 papers were found. After eliminating duplicates, 361 papers remained to be further investigated.

Table 1 Summary of electronic search results

Electronic databases	Query results	After removing duplicates
IEEE Xplore	179	179
ACM Digital Library	17	15
Compendex	86	66
Inspec	83	34
SpringerLink	86	67
Total	451	361

After filtering the results (361 papers) of the electronic search by applying the inclusion/exclusion criteria, we identified 11 papers [5, 26, 35, 43, 48, 49, 68, 80, 94, 97, 106] to include. The manual search of journals and conference proceedings yielded an additional six primary studies (i.e., [16, 32, 34, 41, 92, 95]), and two of them (i.e., [34, 95]) are more recent discussions of and therefore replaced two of the 11 papers identified from the electronic search (i.e., [35, 94]). We scanned the references and the authors’ publication lists of all the 15 primary studies (11+6-2). Five new papers were identified (i.e., [63, 67, 88, 89, 98]), and one of them (i.e., [98]) (with a more complete description

of the approach) replaced one of the already identified 15 primary studies (i.e., [97]). Among all these 19 primary studies (15+5-1), three groups of papers (i.e., [63, 98], [34, 67, 88], and [89, 95]) describe three individual approaches. The papers in each group together describe a single approach. Therefore, eventually a total of 15 approaches (19 primary studies) were included in the review (i.e., [5, 16, 26, 32, 34, 41, 43, 48, 49, 63, 67, 68, 80, 88, 89, 92, 95, 98, 106]). In addition to these 15 approaches (19 primary studies), we also included Abbott's heuristics [3] as one of the primary studies, as discussed in Section 2.3.3. In total, we included 16 approaches (20 primary studies).

2.4 Conceptual framework

We designed a conceptual framework to extract and synthesize data from the primary studies in a systematic and precise way. The conceptual framework is composed of a static model describing common concepts and their relationships (Section 2.4.1), five taxonomies classifying and specifying existing work according to five aspects, specifically the kinds of requirements, the rules imposed on requirements, the types of analysis models, requirements pre-processing approaches, and requirements transformation approaches (Section 2.4.2), and a general transformation process model (Section 2.4.3). This framework defines the common concepts and terminology needed for analysis, synthesis, and comparison of the primary studies. This is paramount since, for instance, different approaches may apply the same techniques but refer to them using different names. The framework therefore provides a way to unify the description of related works. The comparison and evaluation criteria are derived from this framework (Section 2.5).

2.4.1 Static model

In this section, we formalize the notions of transformation, traceability link, requirement, and analysis model by means of a metamodel. The metamodel is presented using the class diagram in Figure 1. It illustrates the main concepts of our review framework and their relationships.

As shown in Figure 1, Requirements can be transformed into an Analysis model either directly or indirectly. For direct transformation, there is no Intermediate Model, so only one Transformation is required: its source is the requirements and its target is the analysis model. For indirect transformation, one or more Intermediate Models¹ are used to bridge the gap between the requirements and the analysis model. Intermediate models function as a temporary source or target of the transformations, which are either from the requirements (source) to the first intermediate models (temp target), between two different intermediate models (one is temp source and the other is temp target), or from the last intermediate model (temp source) to the analysis model (target).

Requirements are composed of one or more Constructs², while an Analysis Model is composed of one or more Model Elements³. Instances of Traceability Link are established between the constructs of the requirements and the model elements of the analysis model. For transformation-based traceability establishment, creating a traceability link is caused by a transformation. When traceability links can be established between a series of models (in the case of intermediate model(s)), we must derive traceability links between the constructs of the requirements and the model elements of the analysis model; these links being modeled as instances of class Derived Traceability Link.

Since requirements are textual specifications, they usually need to be pre-processed either manually or automatically before they are inputted to the transformation. One or more Requirement Pre-processing steps may be taken to transform Requirements into a Pre-processed Requirements, which is further transformed into either an analysis model or an intermediate model if one exists. During a series of transformations, traceability links should be established for the source and target of each transformation, for example, between the requirements and the pre-processed requirements, between the pre-processed

¹ If multiple intermediate models are used, they are ordered.

² A construct can be a sentence, or an actor, if requirements are presented as use cases, for example. We do not distinguish different constructs in this conceptual static model.

³ If the analysis model is presented as a UML model, then model elements are UML model elements. Other representations can be used equally.

requirements and the first intermediate model, between intermediate models if more than one exists, and/or between the last intermediate model and the analysis model.

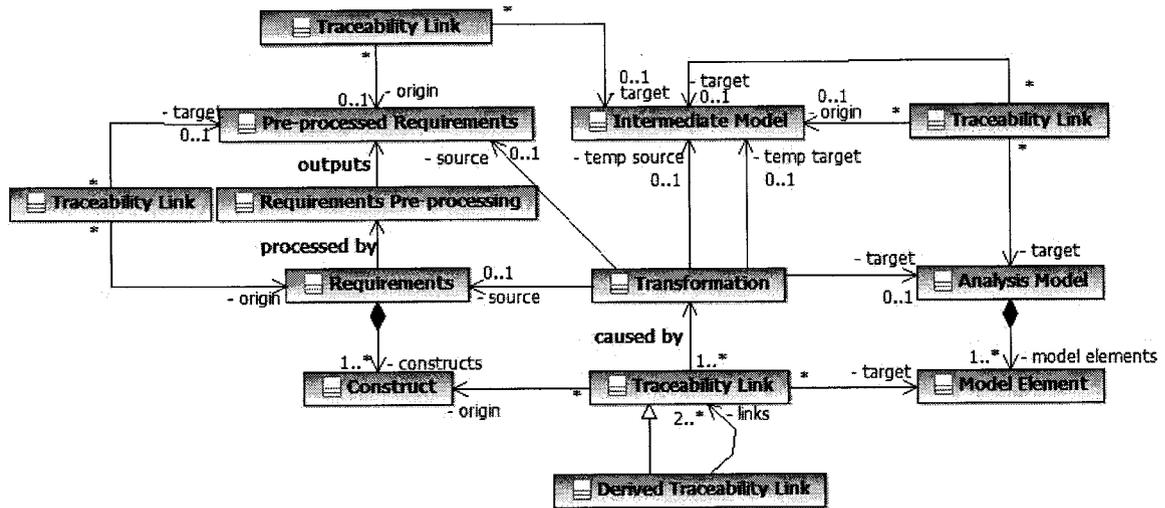


Figure 1 Static model (class Traceability Link appears four times for layout purposes)

2.4.2 Taxonomies

In this section, we define taxonomies to classify and specify the important techniques and terminology used in the primary studies. In the later sections of this paper, these taxonomies will be referred to in multiple places.

The taxonomy of requirements (Section 2.4.2.1) classifies different requirements representations, domain specific information, and whether restricted Natural Language (NL) is applied. A restricted NL is a subset of a natural language, used to restrict its grammar and vocabulary, mostly for the purpose of reducing or eliminating ambiguity and complexity in its usage. The taxonomy of restriction rules (Section 2.4.2.2) classifies different types of restriction rules used for requirements written in restricted NL. The taxonomy of analysis models (Section 2.4.2.3) unifies different analysis models. We also provide a taxonomy of requirement pre-processing approaches in Section 2.4.2.4 to distinguish them at a certain level of abstraction. Last, a taxonomy of approaches for (pre-processed) requirements transformation is presented in Section 2.4.2.5.

2.4.2.1 Taxonomy of requirements

In order to generate analysis models from requirements and further establish traceability links between them, it is important to understand requirements from the following three aspects: which kinds of requirements supplements⁴ are required (sub-taxonomy Domain Specification Information: DSI in Figure 2), how requirements are represented (sub-taxonomy Representation in Figure 2), and whether restricted NL is used when specifying requirements (sub-taxonomy Natural Language in Figure 2). These sub-taxonomies, discussed in the following sub-sections, therefore represent what kinds of requirements are encountered in the literature.

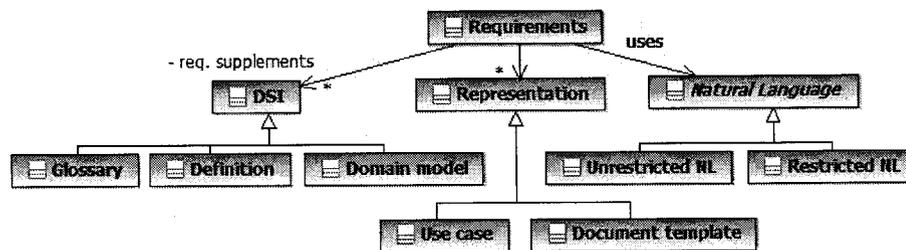


Figure 2 Taxonomy of requirements

Requirements representation

In many situations, requirements are represented as Use cases. It is also possible that a customized Document template is applied to document requirements. Requirements can also be represented using more than one such representation. If no representation is used, then requirements are simply expressed in unstructured natural language. A use case is “the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system. [109]. A use case represents an interaction between a primary actor and other actors, and the system. This interaction is presented as sequences of simple steps (also called flow of events). Use cases are documented following a use case template. There is no standard template, and users typically choose the template that works for them, or is required by a project or a CASE tool. Some companies and organizations rather apply their own

⁴ Requirements supplements refer to documents that clarify the terminology used in requirements.

Document Templates for requirements documentation. These document templates are customized for special purposes such as facilitating requirements elicitation.

Requirements supplements (DSI)

Domain specific information is a necessary input for some of the approaches used to transform requirements into analysis models. It is either captured using a Glossary, Definition, and/or Domain model. A Glossary describes and classifies all the domain-specific terms used in requirements. A Definition [5] defines the notational short hand for expressing requirements in a succinct, practical, and domain-specific way. A Domain model is created to document the key concepts and the vocabulary of an application domain. It describes the various concepts involved in the application domain and their relationships. It is usually represented as a class diagram with possibly constraints in the Object Constraint Language (OCL) [76].

Natural Language (NL)

Requirements can be written using either an Unrestricted NL or a Restricted NL. A restricted NL is also called a controlled NL. It is a subset of natural language obtained by restricting the grammar and vocabulary. It aims to reduce ambiguity, redundancy, size and complexity of requirements, and to facilitate automated analysis.

2.4.2.2 Taxonomy of restriction rules

As shown in Figure 3, we classify the Restriction Rules of the Restricted NLS used in the literature into three types: Sentence Restriction, Sentence Structure Restriction, and Wording Restriction. A sentence is a group of words that are put together to mean something [2], and it is expected to have a subject and a verb. For example, the restrictions on allowed choices of tenses of a verb, on choices of singular or plural forms of a noun are thought of as sentence restrictions. “Use active voice rather than passive voice” is another example of such a restriction. Sentence structure restrictions put restrictions on the structure of a compound sentence. A compound sentence has many clauses. These clauses are joined together with conjunctions, punctuation, or both [2]. For example, “only if-then structure is allowed to describe conditional sentence” is a

restriction on sentence structure. Wording restrictions restrict the choice of words and the way in which they are used: e.g., “use only keywords be or become to express a generalization relationship between the subject and the object of a sentence”.

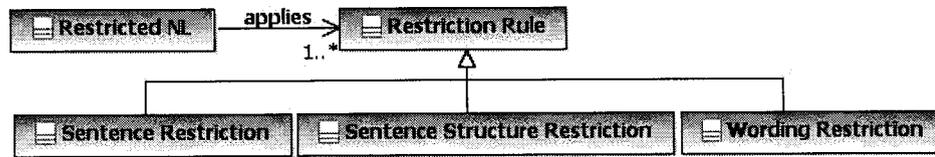


Figure 3 Taxonomy of restriction rules

2.4.2.3 Taxonomy of analysis models

An analysis model is typically presented as a UML model, but not necessarily limited to it. A complete analysis model should describe two aspects of a system: Structure and Behavior. The structure (or static) aspect emphasizes the static structure of the system using classes, objects, attributes, operations, relationships, etc.; while the behavior (or dynamic) aspect emphasizes the dynamic behavior of the system by showing interactions among objects, internal state changes, etc. As shown in Figure 4, we classify the different presentations of the Structure aspect used in the literature into four types: Class Diagram, Object Diagram, Entity Relationship Model (ERM), and Architecture Concept. As two types of UML diagrams, class diagrams are composed of classes, attributes, operations, and relationships among the classes, and object diagrams describe objects and links. ERM was proposed in the early 70’s to document the concepts of entity, relationship, types, and roles. Architecture Concept is used in [41] to present the concepts of components, connectors, and architectural patterns (e.g., client-server). The Behavior aspect is classified into five types: Sequence Diagram, State Machine Diagram, Activity Diagram, Data Flow Graph (DFG), and Message Sequence Chart (MSC). Sequence, state machine, and activity diagrams are three commonly-used UML diagrams for describing the behavior of a system from three different views; sequence diagrams describe object interactions as messages, activity diagrams show the overall flow of control, and state machine diagrams describe state-based behavior. Message sequence charts are very similar to sequence diagrams. Data flow graphs represent data dependencies between operations.

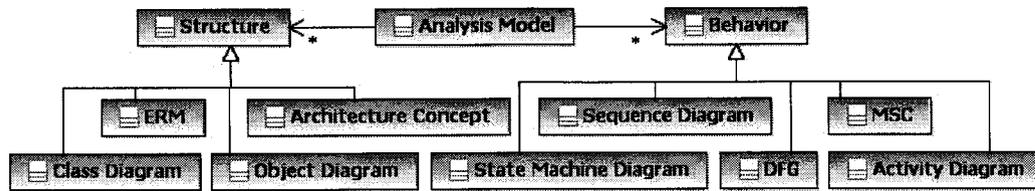


Figure 4 Taxonomy of analysis models

2.4.2.4 Taxonomy of requirements pre-processing approaches

Most requirements are textual and have to be pre-processed (using NL processing techniques) before being used as the input for the next step's transformation. There are usually five types of pre-processing techniques (Figure 5) that can be used in isolation or combined: Lexical Analysis, Syntactic Analysis, Semantic Analysis, Categorization, and Pragmatic Analysis.

Lexical Analysis, also called token generation, is the process of converting a sequence of characters into a sequence of tokens [109]. It is composed of the following processing steps: tokenization, sentence splitting, part-of-speech (POS) tagging, and morphological analysis. *Tokenization* is used to separate words and punctuation, and identify numbers. *Sentence splitting* identifies sentence boundaries within a given text. *POS tagging* identifies words as nouns, verbs, adjectives, etc. *Morphological analysis* returns the root and suffix of each word. Syntactic analysis, also called syntactic parsing, is the process of analyzing a sequence of tokens to determine grammatical structure with respect to a given formal grammar [1]. The output is usually a syntactic parse tree. Semantic analysis is the process of adding semantic information to a parse tree [1], typically by using domain specific information (i.e., DSI). Categorization is the process of recognizing, differentiating, and classifying requirements for some specific purpose, and is usually performed manually. Pragmatic analysis eliminates ambiguities and inconsistencies in requirements. For instance, pragmatic analysis can be used to check the consistency of a new piece of information before it is actually added to existing requirements [68].

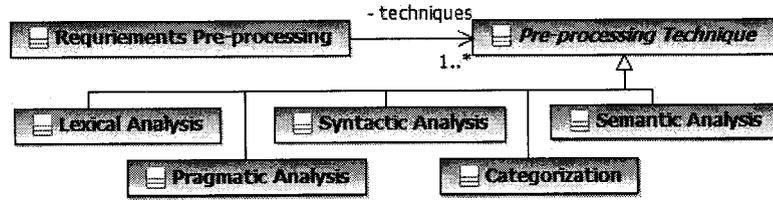


Figure 5 Taxonomy of requirements preprocessing approaches

2.4.2.5 Taxonomy of transformation approaches

Pre-processed requirements are further transformed into an analysis model or an intermediate model. Three types of transformations can be identified: Rule based, (the most commonly used in the literature) Ontology based, and Identity Transformation (Figure 6). Rule based transformation utilizes a set of predefined Transformation Rules. An ontology is a “shared vocabularies for describing the relevant notions of a certain application area, whose semantics is specified in a (reasonably) unambiguous and machine-processable form” [12]. An ontology model is built when NL sentences are processed. This ontology model acts as an intermediate model that is further transformed into an analysis model. Such transformations are called Ontology based transformations. An Identity Transformation transforms a model (source) into another model (target) without change in information content: the two models describe the same concepts but with different representations. A Pattern based transformation transforms source patterns into target patterns. A source pattern describes and organizes a set of source elements; while a target pattern describes and organizes a set of target elements.

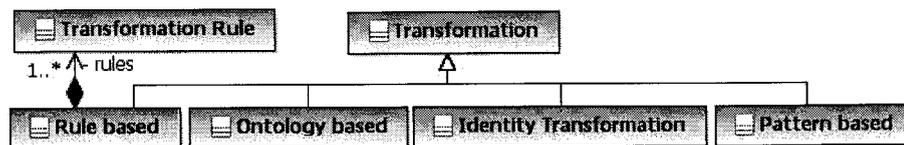


Figure 6 Taxonomy of transformation approaches

2.4.3 Process model

We use an activity diagram (Figure 7) to model the overall process of transforming requirements into an analysis model. First, requirements are pre-processed by applying one or more pre-processing techniques (Section 2.4.2.4), resulting into pre-processed

requirements (step 1). If there is no intermediate model, then the pre-processed requirements are transformed directly into an analysis model (step 6); otherwise, the pre-processed requirements are transformed into an intermediate model (step 2). If there is more than one intermediate model involved, then transformations between these intermediate models are performed (step 3). Step 3 can be performed more than once, depending on the number of intermediate models. For example, if there are three intermediate models, step 3 is performed twice. Then step 4 transforms the last intermediate model into the analysis model. While steps 2, 3, and 4 (or 6 if no intermediate model is used) are performed, traceability links are established between the source model and the target model of transformations (step 5). The output of this step is several sets of traceability links either between the requirements and the first intermediate model, between two intermediate models, or between the last intermediate model and the analysis model. Finally, we derive traceability links for the requirements and the generated analysis model (from step 4) from the sets of traceability links involving intermediate models (step 7). If there is no intermediate model (i.e., step 6 is taken), step 7 is obviously not needed. The output of the whole process is an analysis model, and a set of traceability links between the requirements and the generated analysis model.

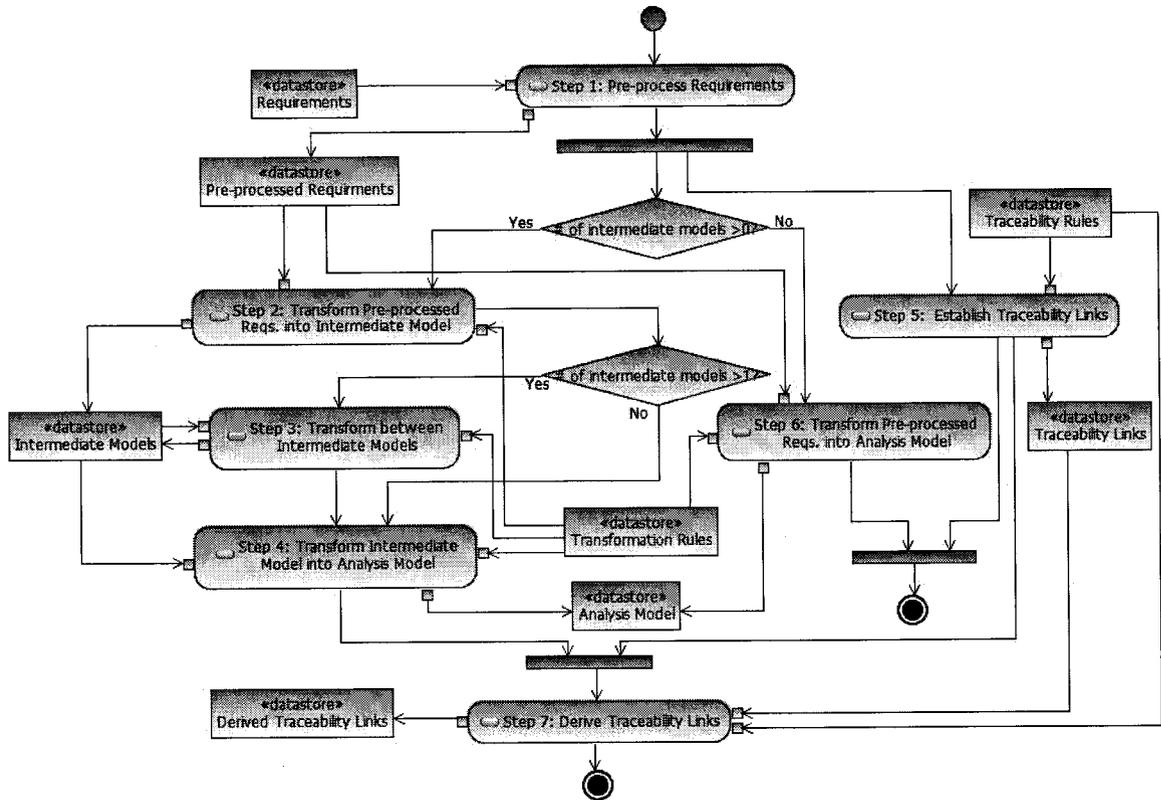


Figure 7 A generic transformation process

2.5 Evaluation Criteria

Our evaluation criteria are derived from the conceptual framework discussed in Section 2.4. Before specifying them, we first clarify their mapping to the conceptual framework (Figure 8). As we have discussed in Section 2.4, the conceptual framework is composed of a Static model, five Taxonomies and one Process model. The evaluation criteria, being discussed in this section, are used to evaluate each reviewed approach in terms of their inputs (i.e., Difficulty of documenting requirements), their outputs (i.e., Completeness of analysis models), and their transformation approach from the following six aspects: Automation, Efficiency, Evaluation, Traceability capability, Structuredness of transformation rules, and Completeness of transformation rules. As shown in Figure 8, for example, the evaluation criterion Difficulty of documenting requirements (Section 2.5.1) is derived from the Taxonomy of requirements (Section 2.4.2.1). Notice that the Static model of the conceptual framework (Figure 1) formalizes a number of basic notions such as transformations and requirements. The static model is not directly related

to any evaluation criteria; however the taxonomies and the process model are all dependent on it. Last, note that the criterion Evaluation methods in primary studies reports, for example, on the number and size of the case studies performed and it is not therefore traced back to the conceptual framework.

The conceptual framework is used to extract and synthesize data from the primary studies in a systematic and precise way and then these data, presented as a table (Table 2), is analyzed according to the evaluation criteria, leading to the evaluation results reported in Section 2.6, in which we also summarize the restriction and transformation rules used in the approaches. As shown in Figure 8, these two summaries are traced back to the taxonomy of restriction rules and the taxonomy of transformation approaches, respectively.

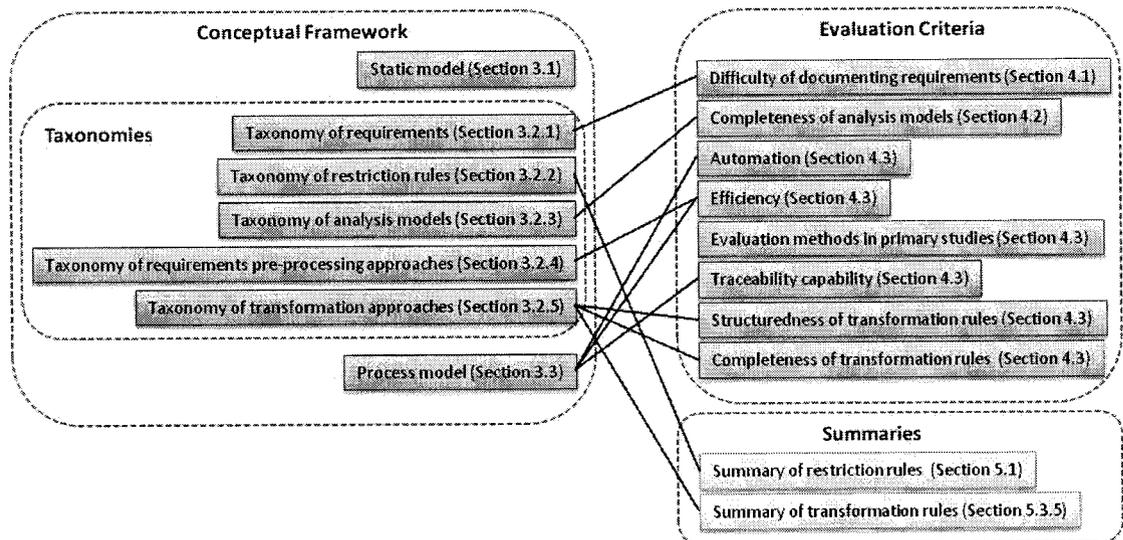


Figure 8 Mapping between the conceptual framework and evaluation criteria and summaries

2.5.1 Evaluation criterion for requirements

We need to assess how difficult it is to document requirements in the format required by a specific approach. We do so by considering whether any DSI (i.e., Glossary, Definition, and Domain Model) is required, whether a restricted NL is enforced to write requirements, and whether the requirements representation is commonly used and well-supported in practice. If an approach requires DSI, a restricted NL is enforced, and the

requirements are represented using a specific template (e.g., not standard or commonly used), documenting requirements is deemed difficult. At the other end of the spectrum, if an approach does not require DSI, applies unrestricted NL, and applies commonly used requirements representations (e.g., use cases descriptions), then requirements are deemed easy to document.

2.5.2 Evaluation criterion for analysis models

We need to evaluate the generated analysis models with respect to their completeness. From a user's perspective, a generated analysis model is expected to be as complete as possible so that it can be a useful starting point of an iterative analysis refinement process. Various types of analysis models can be generated by the approaches proposed in the primary studies, such as UML models [106] and MSCs [31]. Though it is difficult to compare different types of analysis models, their common modeling capabilities can be extracted, and used as the basis for comparison. For example, MSCs can be considered similar to UML sequence diagrams, because both model the dynamic behavior of the system in terms of object interactions through messages. If a generated analysis model both describes the system structure (e.g., class diagram) and behavior (e.g., sequence diagram, state machines, or activity diagrams), then we label the generated analysis model as complete. If a generated analysis model describes only one of these two aspects of a system (i.e., either the structure or behavior), then we label it as incomplete.

2.5.3 Evaluation criteria for transformation

We evaluate the transformation approaches proposed in the reviewed approaches with respect to their automation and efficiency. The automation criterion evaluates whether a transformation is automated, automatable, semi-automated, or manual. A transformation approach is automated if it has been fully implemented. If a transformation algorithm is proposed in a paper, then we assess whether we deem the description is sufficient to implement it, and if this is the case, the transformation approach is deemed automatable. In some cases, a transformation is semi-automated because user interventions are required. Last, some approaches are entirely manual.

The efficiency of an approach is evaluated by analyzing how many transformation steps are necessary, and how many requirements pre-processing techniques (Section 2.4.2.3) are applied. If it takes several transformation steps for an approach to transform requirements into an analysis model, we label this approach's efficiency as low, as opposed to an approach requiring only one single step. If an approach needs three or more requirements pre-processing techniques, we also label it as having a low efficiency. Although other measures of efficiency could be considered (e.g., execution time), our measure of efficiency was deemed sufficient since a number of the approaches we compared are not fully-automated. We also observed that the larger the number of steps, the larger the proportion of manual steps. Therefore an approach with a high number of steps (using our measure) requires more (manual) effort, and we refer to this situation as a low efficiency.

Extensive case studies are a necessity since validating transformation approaches cannot be performed in an analytical way. Selecting case studies to run and how results are analyzed, are two important aspects of the evaluation of an approach. We evaluate each approach and examine: 1) the number and size of the case studies performed, 2) the results of the case studies reported, and 3) whether other evaluation approaches (besides case studies) are described.

Traceability links between requirements and analysis model elements are expected to be established when a transformation is performed. Because only a few of the reviewed approaches report on traceability, we only examine whether or not traceability is reported in each approach and do not analyze the details of the traceability link generation strategies.

Transformation rules specify which requirements constructs map to which analysis model elements. They are expected to be complete and well-structured. If, according to our understanding to the transformation rules as they are described in primary study papers, the transformation rules proposed in an approach can transform most or all requirements constructs into analysis models elements, then we say this set of transformation rules is complete; otherwise, incomplete. We expect each approach to evaluate the completeness

of its transformation rules by for example, performing case studies. However some of the approaches do not evaluate the completeness of transformation rules and some of them don't even describe transformation rules. This simply makes the evaluation impossible. If transformation rules are presented in the primary studies and organized according to the structure of the source language, the target language, some other relevant organization (e.g., rule composition), or different transformation phases [23], and each transformation rule is well specified (e.g., using a carefully defined language like OCL), then we label the transformation rules as well-structured.

2.5.4 Discussion

One may argue that it is possible to perform a finer-grained, more objective analysis such as evaluating how restrictive each restriction rule is if restricted NL is applied, how complete is each aspect (or diagram) of generated analysis models (e.g., amount of information generated in the class diagram), how efficient is each pre-processing technique and each transformation step. However, it is difficult (if not impossible) to perform such an analysis because: 1) No sufficient information is provided in the primary studies (e.g., in many cases no case study is presented and the completeness of their transformation rules is not discussed, when they are described in detail); 2) Empirical studies are needed to perform a finer-grained analysis to evaluate the restrictiveness of each restriction rule and the overall completeness of each diagram, which is out of the scope of this paper; 3) Some approaches are manual; therefore the completeness and correctness of generated analysis models and their overall efficiency strongly depend on the capability of users; 4) Different types of diagrams are generated and therefore it is difficult to have common evaluation criteria for evaluating the completeness of generated analysis models; 5) It is common for primary studies to use different case study systems and therefore it is hard to have objective evaluation criteria for the completeness of the generated analysis models.

Our evaluation criteria, though coarse-grained, are still sufficient to differentiate each approach and are straightforward to apply, thanks to the well-specified conceptual framework and the clear mapping between it and the evaluation criteria. Furthermore, as

illustrated by the results of the comparison, no such fine-grained analysis was required to compare approaches: our criteria are precise enough to allow us to differentiate different approaches.

2.6 Synthesis and Evaluation

In Section 2.4.2 we defined taxonomies to classify and characterize techniques and concepts used in the primary studies. The selection of one or more than one element from each of these taxonomies is denoted as a *configuration* characterizing a given approach. Such configurations are a way for us to abstract away from details and allows the analysis of emerging, general patterns. The taxonomy of requirements contains three sub-taxonomies: DSI, Representation, and Natural Language. A combination of one or more element from these three sub-taxonomies forms a *requirements configuration*. For example, if an approach does not require DSI, is based on use cases which are described in restricted NL, then the requirements configuration of the approach is presented as a tuple $(None, Use\ Case, Yes)$. As shown in Table 2, the approach proposed in [92] conforms to this requirements configuration (configuration 5 in Table 2, Column 2). Steps taken by each approach are different. For example, some approaches (e.g., [80]) that do not contain intermediate models but require requirements pre-processing contain only Step 1 and Step 6 (Figure 7), presented as a tuple $(1, 6)$ in Table 2, Column 5. If an approach contains two transformations (one intermediate model), the transformation from pre-processed requirements to the intermediate model is rule-based, and the transformation from the intermediate model to analysis models is also rule-based, we use a tuple (R, R) to represent the configuration of the transformations, as shown in Table 2, Column 6. Over all, an *approach configuration* is characterized by a requirements configuration, analysis models, requirements pre-processing techniques, steps taken by the approach, types of transformations, and automation and efficiency of the approach. Configurations of each approach are given in Table 2, grouped by requirements configurations.

Table 2 Evaluation summary of the approaches proposed in the primary studies

Requirements Configuration ⁵		Analysis Model ⁶	Steps	Transformation ⁸	Automation	Efficiency	Primary studies
#	(DSI, Representation, Restr. NL?)						
1	(None, None, No)	Object diagrams Classes, attributes, and associations Domain models, hybrid activity diagrams Class, activity, state machine diagrams Architecture concepts (e.g., components and connectors). Class diagrams Class diagram, coarse-grained behavioral concept Data types, variables, operations, control constructs (e.g. <i>if-then-else</i> and <i>for loop</i>) ⁹ ERM, DFG, UML models (not described) Class diagrams Sequence diagrams Message sequence charts Class and sequence diagrams Extended sequence and extended activity diagrams ¹⁰ Class diagrams State machines	LA, SynP, (1,2,4) SemP, PA LA, SynP, SemP (1,2,4) LA, SynP (1,2,3,4) LA, SynP, SemP (1,2,4) Catg (1,2,4,5) LA, SynP (1,6) Catg (1,2,4) None (6) LA, SynP, SemP (1,2,3,4) LA, SynP, Catg (1,6) Unknown (? , ? , 4) Catg, LA, SynP (1,6) None (5,6) None (6) LA (1,2,3,4) None (6)	(R, R) (O, R) (R, R, R) (R, R) (R, R) (R, R) (R) (R, P) None (I, R, R) (R) (?, P) (R) N/A (P)	Automated Automated Automatable Semi-automated Manual Semi-automated Manual Manual Automated Automatable Automated Automated Manual N/A Automated Automated	Low Low Low Low N/A N/A N/A N/A Low Low Unknown Low N/A N/A Low Low High	[68] [43] [48] [34, 67, 88] [41] [80] [16] [3] [5] [106] [26] [31] [49] [92] [63, 98] [89, 95]

⁵ The requirements configurations requiring more significant effort to document requirements are highlighted with a darker color.

⁶ The approaches that are capable of generating analysis models, i.e., including both structural and behavioral aspects of a system, are highlighted.

⁷ *LA* denotes Lexical Analysis; *SynP* denotes Syntactic Parsing; *SemP* denotes Semantic Parsing; *Catg* denotes Categorization; *PA* denotes Pragmatic Analysis; *None* means that the corresponding approach does not need requirements pre-processing.

⁸ *R* denotes rule based transformation; *O* denotes ontology based transformation; *P* denotes pattern based transformation; *I* denotes identity transformation.

⁹ The heuristics of [3] have been adapted in Software Engineering textbooks (e.g., [14, 61]) to generate UML class diagrams as analysis models.

¹⁰ The approach extends UML sequence and activity diagrams to represent requirements including some concepts of use case models (e.g., precondition).

2.6.1 Requirements configurations

A total of seven different configurations match the reviewed approaches (Table 2, Column 1). Configuration 1 requires no DSI, no specific representation, and no restricted NL. This configuration is the most frequently used one; eight out of 16 approaches comply with this configuration. Configurations 2, 6 and 7 require a DSI (Section 2.4.2.1) as part of their requirements input, to assist the computational NL processing. For example, a glossary is mainly used to identify entities, objects, or classes. A domain model serves as the structural basis of target models such as sequence diagrams. Most of the domain specific information is manually constructed. The rest of the configurations do not require any DSI.

Configuration 3 needs requirements to be documented using the OBFS template, a customized document template that the transformation technique of the approach [106] relies on. This configuration does not need any DSI or restricted NL. Configurations 4-7 (six approaches) take use cases as requirements representation. This is reasonable since use cases are a commonly used notation for capturing requirements in practice. Besides, a use case template helps organize textual requirements so that the requirements pre-processing and the following transformation(s) can be facilitated.

Configurations 3, 5-7 require that requirements be documented using restricted NL. There are three main reasons why restricted NL is used in requirements documentation. A restricted NL aims to reduce ambiguity, redundancy, and complexity in documents. It also makes computational NL processing more reliable, efficient, and accurate. Last, it facilitates translation into other languages. However the extent of restrictions varies across approaches and a balance should be struck between the applicability of restriction rules and facilitating analysis. We summarize the restriction rules applied in the primary studies (i.e., [92, 95, 98, 106]) in Table 3. These rules are classified into three categories: sentence restrictions, sentence structure restrictions, and wording restrictions (Section 2.4.2.2). The table also indicates where rules are applicable and their purpose. Examples are given for some rules. The restricted NL used in [92] is not well described in the paper. That is why only one restriction rule is presented in the table.

Based on the data we extracted from each primary study and summarized in the first column of Table 2, we discuss next how difficult it is to document requirements in the format required by a specific approach according to our evaluation criteria (Section 2.5.1). The configurations requiring more significant effort to document requirements are highlighted with a darker color, following the rationale described next. The evaluation results show that it is most difficult to document requirements in the format required by configuration 7 (approach proposed in [89, 95]) because a great deal of user effort is needed to obtain a domain model containing classes, associations, and operations, which are indispensable for generating state machines, and additionally use cases are required to be written in restricted NL. Configurations 2, 3, and 6 require the second largest effort to document requirements. Though configuration 2 [5] does not rely on restricted NL and does not require any specific requirement representation, the difficulty of documenting requirements is still high as users are required to manually specify glossary and a significant number of definitions in a specific form. Configuration 3 [106] implies requirements to be manually documented in a non-standard modeling language (OBFS) and the use of restricted NL. Configuration 6 [63, 98] needs a glossary, and use cases are required to be written in restricted NL. Configuration 5 requires even less user effort since no DSI is necessary. Configuration 4 requires less effort than Configuration 5 to document requirements as not only no DSI is required but additionally use cases do not need to be documented using restricted NL. Configuration 1 requires the least effort to document requirements.

2.6.2 Analysis models

We can see from Table 2, Column 2 that twelve out of 16 approaches can derive structural model elements (e.g., objects, classes, associations, components) from requirements. Most of the approaches are able to generate objects, classes, and associations, but not all of them can generate attributes, operations, and generalizations. Nine approaches can generate behavioral features of a system (e.g., sequence diagrams, state machines, and/or activity diagrams).

Three approaches ([48], [34, 67, 88] and [16]) (highlighted) conforming to configuration 1 are capable of generating analysis models including both structural and behavioral aspects of a system, which are characterized as complete according to our evaluation criteria (Section 4.2); two approaches ([5] and [49]) (highlighted) conforming to configuration 2 and 4, respectively, can also generate complete analysis models. The generated domain models of the approach proposed in [48], conforming to configuration 1, contain only objects and links, rather than commonly-used class diagram representations; the generated hybrid activity diagrams (i.e., UML activity diagrams also including the concepts of actors, business rules, and messages) are at a very high level of abstraction, and are independently generated from the generated domain models (i.e., there might be inconsistencies between the two diagrams). The NIBA project [34, 67, 88], also conforming to configuration 1, can derive class, activity, and state machine diagrams from requirements. User intervention is required in many places, especially during the transformation from requirements to intermediate models. There is not enough information provided in the papers to show that the generated class, activity, and state machine diagrams are correct, consistent, or precise enough. The approaches proposed in [16] and [49], conforming to configurations 1 and 4, are all manual; therefore the completeness and correctness of generated analysis models mainly depend on the capability of users, rather than the approaches themselves. The approach proposed in [5] requires a great deal of user effort on documenting requirements, two intermediate models (three transformations), and a sequence of requirements pre-processing techniques.

Not surprisingly, UML (e.g., class, activity, sequence, and state machine diagrams) is the most frequently used language in the reviewed approaches to represent generated analysis models.

2.6.3 Transformation - automation

Only five approaches describe the algorithms they used to various extents of details. Most of these algorithms are not described at a level of detail that is amenable to an implementation. According to the evaluation criterion discussed in Section 2.5.3, we

summarize the evaluation results of transformations: automated, automatable, semi-automated, or manual.

As shown in Table 2, Column 7, seven out of 16 approaches are automated; two are not automated but are automatable; two approaches require user intervention to semi-automatically perform the transformation; four approaches require manual transformations. Complex pre-processing techniques are required for all the automated approaches, except the approach proposed in [63, 98] (Configuration 6), which only requires lexical analysis, and the approach proposed in [89, 95] (Configuration 7), which does not have any requirements pre-processing techniques since the transformation from use cases plus a domain model to state machines relies on the template structure of the use cases and the domain model. However two intermediate models (three transformations) are required in this approach (Column 5). For the approach proposed in [26] (first approach in Configuration 4), the transformation from use cases to intermediate models (Step 2) is not described in the paper and therefore the automation of this step is unknown as indicated in the table. The approaches proposed in [89, 95] and [31] have been implemented and therefore they are automated approaches. The one proposed in [106] is not automated but is automatable and the one proposed in [80] is semi-automated since a significant user intervention is required. The transformation is not explicitly discussed in [92], because the approach does not attempt to provide a solution for the transformation of requirements into analysis models though the proposed approach can be adapted to that purpose, which is also the reason why we included this paper for review. Last, three manual approaches are proposed in [49], [16] and [3] respectively. Though the approach proposed in [89, 95] is automated, a great deal of user effort is needed to obtain a domain model and specifying use cases and applying restrictions. Additionally, the consistency between the domain model and the use cases must be manually maintained. Manual requirements pre-processing (e.g., users are required to manually classify the sentences) is required for the automated approach proposed in [31].

2.6.4 Transformation - efficiency

As we have discussed in Section 2.5.3, the efficiency of an approach is evaluated by analyzing how many transformation steps are taken in the approach, and how many requirements pre-processing techniques are applied.

As shown in Table 2, Column 4, most approaches apply at least one of the requirements pre-processing techniques (Section 2.4.2.3). We don't know what requirements pre-processing techniques are applied in the approach proposed in [26], since it is not described in the paper. The approach proposed in [89, 95] does not have any requirements preprocessing technique since the transformation from use cases plus a domain model to state machines relies on the template structure of the use cases and the domain model. The approach proposed in [92] does not require any requirements pre-processing technique because the approach describes three equivalent requirements representations, and each of them can be transformed into the other. The approach proposed in [49] does not need any requirements pre-processing technique since it proposes a set of techniques for users to manually specify requirements and also a process to guide the users to derive the conceptual models from the requirements. It does not aim to automatically transform requirements into an analysis model. Similarly, Abbott's heuristics [3] do not need any requirements pre-processing technique.

As shown in Table 2, Columns 5 and 6, rule-based transformations are most frequently used to create the first intermediate model (Step 2 of the process): first letter in the transformation tuple (Column 6). An ontology-based transformation is used in [43] since the intermediate model is all ontology-based. A "?" for approach [26] indicates that the transformation is unknown since it is not discussed in the paper. Only one approach [92] applies pattern-based transformations (denoted as "P") and only one approach [5] applies identity transformation (denoted as "I"). Most of the approaches containing Step 4 use rule-based transformations (except [16] and [26]). Eight approaches use intermediate models (containing Step 2), when direct transformation from requirements to an analysis model cannot be achieved. Two intermediate models (three transformations, and therefore Step 3 is required) are contained in [5], [48], and [63, 98] instead of only one

intermediate model (two transformations) in the other six approaches that use intermediate models. Most of the approaches use rule-based transformations to transform pre-processed requirements directly into an analysis model (Step 6).

According to our evaluation criterion on efficiency of approaches, the approach proposed in [89, 95] shows highest efficiency because it does not need any requirements pre-processing technique and requirements are directly transformed into analysis models. Note that it does not make sense to evaluate the efficiency of manual approaches so their efficiency is marked as “N/A”.

2.6.5 Transformation – others

2.6.5.1 Evaluation

Only four out of 16 approaches have their transformation approaches evaluated. Case studies have been performed to evaluate the approaches proposed in [43] and [26] by manually comparing the tools results with the manually constructed analysis models. A performance evaluation method is also proposed in [43] and five case studies were performed to evaluate the performance of the tool. The evaluation results show that the approach can perform better than other language processing technologies, such as information retrieval systems. Three industrial pilot studies were performed to test the acceptability of the tool implementing the approach proposed in [5]. The evaluation of the approach proposed in [34, 67, 88] is not discussed in details in the papers, except for the statement that “the approach has been applied for practical requirements analyses and the results showed to be encouraging.” The other approaches were not evaluated, though some of them present a running example to illustrate their approach rather than to evaluate it.

2.6.5.2 Traceability support

Among the papers we have reviewed, only two transformation approaches [41] and [49] report on traceability. In [41], it is claimed that traceability is supported, though this is not discussed in the paper. A traceability model, represented as a function decomposition table (rows are use cases and columns are the identified classes), is proposed in [49] to

link the identified classes to the use cases. Deriving traceability links (Step 7) from already established links is not an issue for transformation approaches that do not involve intermediate models; however for those which require one or more intermediate models it is an indispensable step since from the users' perspective it is very important to access derived traceability links between requirements and analysis models without having to deal with the intermediate model(s). This step is not covered in any of the approaches we reviewed.

2.6.5.3 Completeness and structuredness of transformation rules

Nine ([68], [43], [63, 98], [48], [34, 67, 88], [16], [31], [106], and [3]) out of 16 approaches describe their transformation rules in their primary studies but none of them evaluate the completeness of the transformation rules. Five ([5], [26], [41], [89, 95] and [80]) out of 16 approaches do not describe their transformation rules at all. Note however that the completeness of the transformation patterns of [26] was evaluated by performing some case studies (not described in the paper though). The evaluation was manually performed by comparing the tool generated interaction models with the ones manually constructed by the experts. The evaluation results show that 65% of the sequence diagram fragments generated by the tool are identical (i.e., modeling the same interactions with the same instances and the same messages) to the manually obtained sequence diagram fragments, 28% of the automatically generated fragments are equivalent (i.e., modeling the same interactions with different instances and messages) to the manually obtained one, and 7% of these fragments are different (modeling different interactions). The approaches proposed in [49] and [92] do not purport to provide solutions for transforming requirements into analysis models; though both of them can be adapted to that purpose, which is also the reason why we included them. Therefore no transformation approach is discussed in these two papers.

Seven approaches directly transform requirements into analysis models (Step 6): [80], [106], [31], [49], [92], [89, 95], and [3]. The others use intermediate models to bridge the gap between requirements and the analysis model. Transformation rules of these indirect transformation approaches contain two rule sets: transformation rules from requirements

to intermediate models and transformation rules from intermediate models to the analysis model. The intermediate models act as the target models of the first rule set and also the source models of the second rule set. Because of the differences among these intermediate models, it is hard to synthesize these rules. Therefore, we only summarize and synthesize the transformation rules of the rule-based transformation approaches that directly transform requirements into an analysis model, except for the approach proposed in [92] in which the transformation is not explicitly discussed, because the approach does not aim to provide a solution for the transformation from requirements into an analysis model though the proposed approach can be adopted to that achieve. The papers [89, 95] and [80] do not describe the transformation rules they used. The transformation rules from [31], [106], [49], and [3] are presented in Table 4 and Table 5. We also summarize (in the same tables) the heuristics rules proposed in [14, 60, 62] which extend or do not refer to Abbott's heuristics rules [3]. Their completeness, effectiveness, and correctness are not evaluated through empirical studies. Though the approach proposed in [43] does not directly transform requirements into an analysis model (intermediate model is required), the paper describes the mapping relations between the two types of transformation rule sets and therefore the mapping relations are derived as transformation rules and also included in the Table 4 and Table 5. In each one of these four approaches, transformation rules are independent from each other: Each rule simply describes the mapping relationship between a requirements concept (Column 2) and an object-oriented concept (Column 3). Requirements constructs include natural language concepts (e.g., noun, subject, etc.). In Column 4, constraints are provided when necessary. Column 6 provides some examples for the transformation rules that are not easy to understand.

2.6.5.4 Summary of evaluation results

An ideal approach for transforming requirements into analysis models would have the following characteristics: 1) requirements should be easy to document using the format required by the approach, 2) generated analysis models should be complete (i.e., contain structural and behavioral aspects of a system), 3) the approach should contain the least number of transformation steps as possible (high efficiency), 4) the approach should be automated, and 5) the approach should support traceability management (Step 5).

However none of the reviewed approaches conforms to the ideal configuration, as described next.

1. Requirements configuration

- a. Requirements configuration 1 (Table 2)

The approaches conforming to requirements configuration 1 require the least user effort to document requirements. However, only two of these approaches are automated and one is automatable. The other five approaches are either semi-automated or completely manual. Besides, complicated requirements pre-processing techniques and intermediate models are required for the two automated approaches and therefore their efficiency is low. It is also worth noticing that these two automated approaches are not capable of generating complete analysis models, i.e., including both static and dynamic aspects of a system.

- b. Requirements configuration 4

Requirements configuration 4 ranks second in terms of user effort to document requirements. Two of the three approaches conforming to this configuration are automated, which however can only automatically generate the behavioral aspect of a system, instead of a complete analysis model.

- c. Requirements configuration 5

Compared with requirements configuration 4, requirements configuration 5 requires use cases to be documented in restricted NL; therefore it requires more user effort to document requirements. The approach conforming to this configuration still cannot generate complete analysis models.

- d. Requirements configurations 2, 3 and 6

Two approaches conforming to requirements configurations 2 and 6, respectively, are automated and the approach conforming to configuration 3 is automatable. Requirements needed by these three configurations rank second in terms of

documentation difficulty. Only one of them (i.e., [5]) is capable of generating a complete analysis model. Additionally the efficiency of the approach is low since two intermediate models (three transformations) and a sequence of requirements pre-processing techniques are needed.

e. Requirements configuration 7

Requirements configuration 7 is the one that requires the most user effort to document requirements. The approach is automated and does not need requirements pre-processing. The efficiency of the approach conforming to this configuration is high. However the approach still cannot generate complete analysis models (i.e., static and dynamic aspects).

f. As expected, approaches requiring more user effort to document requirements achieve better automation and higher efficiency.

g. Use cases are the most frequently applied requirements representation.

2. Analysis model representation

UML diagrams are the most frequently-used representations of analysis models, which confirms that in practice UML is used in many IT software development organizations [81].

3. Efficiency

a. Requirements pre-processing

Most of our reviewed approaches apply at least one of the requirements pre-processing techniques, among which lexical analysis (Section 2.4.2.3) is the most commonly used technique. This is understandable because requirements are usually written in textual form that must be tokenized and POS of sentences should be identified in order to facilitate transformations. Syntactic parsing (Section 2.4.2.3) is also commonly applied in the approaches that require determining grammatical structures such as subjects and predicates of sentences.

When not applying any pre-processing technique (except categorization), one needs to manually transform requirements into intermediate models or analysis models. Categorization (Section 2.4.2.3) is another technique frequently used in the primary studies. All these papers require categorization to be performed manually. Complex pre-processing techniques are usually required for automated approaches.

b. Transformation steps

Most of the reviewed approaches have one intermediate model. Few of them need two intermediate models. For those using intermediate models (containing Step 2), rule-based transformations are most frequently used.

c. Efficiency

According to our evaluation criterion on efficiency of approaches, only one of the reviewed approaches [89, 95] has clearly superior efficiency because it does not need any requirements pre-processing technique and requirements are directly transformed into analysis models.

4. Automation

More than half of the reviewed approaches are automated or automatable. A high level of automation is an absolute requirement for any approach to scale up in industrial practice.

5. Approach configuration

No approach, with acceptable user documentation effort and efficiency (e.g., one or two transformation steps), is currently able to automatically or semi-automatically generate a complete (i.e., containing both static and dynamic aspects), consistent analysis model.

The systematic review results show that no approach is able, based on reasonable requirements definition effort and in an efficient manner, to automatically or semi-

automatically generate a consistent analysis model, which is expected to model both the structure and behavior of the system at a logical level of abstraction. For example, in the context of UML modeling, analysis models should at least contain a class diagram and interaction diagrams for each use case [14]. We identified a number of variation factors in existing approaches leading to different drawbacks and strengths. These factors included (1) whether use cases or other concepts were used to structure requirements, (2) what use case template was used, (3) what NL restrictions, if any, were applied when specifying use cases, (4) whether domain specific information (e.g., glossary, domain model) was expected, (5) if the approach relied on one or more intermediary models between requirements and the analysis model.

After pondering carefully all variation factors, based on existing studies, we selected the following strategy to strike what we thought was the best balance between practicality, level of automation, and required requirements engineering effort: 1) We do not use any domain specific information. 2) We adopt UCMs and a specific template to structure textual requirements. 3) The restrictions we proposed for NL and UCSs are based on our systematic literature review (see next) and appear to be easy to use (Chapter 12). 5) We use UML as the notation for analysis models. This conforms to the MDA [57] transformation concept, which requires the source and target of a transformation be represented as UML models. 6) Only one intermediate model is used. 7) We rely on transformation rules that are well structured and precisely specified, so that they are easy to extend and modify. 8) Our method establishes traceability links during transformations.

We also investigated whether existing approaches to formalize textual UCMs (referred to as use case metamodels) could be the intermediate model of our method. The approaches we have found [25, 70, 78, 92, 102] fail to satisfy our objectives for one or more of the following reasons: A large amount of NL information is not captured by the metamodel; The metamodel does not handle any NL information; No sentence-level concepts can be described with the metamodel. To compare with these existing metamodels, our UCMeta metamodel is more complete and complex and can facilitate our objectives as an intermediate model.

Table 3 Restriction rules on requirements documentation

Restriction	Restriction rules	Applying situation	Purpose	Rel. works
Sentence restriction	Apply simple sentence ¹¹	Any statement	Facilitate automatic NL parsing; reduce ambiguity; simplify the complexity of sentences.	[92, 106]
	Use active voice rather than passive voice (actor is omitted)	Any statement	Facilitate automatic NL parsing; easier to identify messages or behavior.	[98]
	Use the same verb for the same action in different sentences	Use case → flow of events	Improve the quality of NL parsing; reduce ambiguity.	[98]
Sentence structure restriction	Don't use pronouns	Any statement	Facilitate automatic NL parsing	[98]
	and, or	Use case → condition	Specify composite conditions	[95]
	GO TO Step [number]	Use case → Branching statements	Specify branching	[95]
	CON [statement]	Concurrency statements	Specify concurrency statements	[98]
	if-then	Conditional statements	Specify conditional statements	[98, 106]
	While-endWhile, Repeat[number]until[states], Do-until	Iteration statements	Facilitate the transformation to sequence diagram	[98]
	AFTER [duration], BEFORE [duration]	after delay and before delay statements	Facilitate the transformation to state machines (timeout transitions)	[95]
Wording restriction	AND ON [entity]	Use case → Postcondition	Facilitate the transformation to state machines	[95]
	is a kind of, is specialization of, is generalization of	Inheritance sentences	Identify generalization between subject and object	[106]
	drive, work for, maintain, manage, own, execute, serve, use	Action sentences	Identify objects and associations	[106]
	talk to, communicate with, refer to	Communication sentences	Identify objects and associations	[106]
	next to, goto	Location sentences	Identify objects and associations	[106]
	has (a capability of), has (a capacity for), can, able to	Behavioral sentences	Identify behaviors	[106]
	has not (a capability of), has not (a capacity for), cannot, not able to			

¹¹ A simple sentence is composed of one subject and one predicate.

Table 4 Summary of transformation rules (part 1)

Transformation rule		Rel. work	Example
Requirements concepts	OO concepts		
(recurring) noun or noun phrase	object, class	[3, 14, 43, 62]	
subject of a sentence	object, class	[31, 107]	the subject is noun
object of a sentence	object, class	[31, 107]	the object is noun
actor of use cases	object	[14, 49]	
use case	object	[14]	<<control>> object
genitive case (e.g., using <i>of</i> , <i>'s</i>)	attribute	[14, 43]	the first noun is the attribute of the second noun
the object (noun) of a simple sentence	attribute	[43, 106]	the predicate of the sentence contains <i>has</i> <i>consist of</i> <i>contain of</i> <i>denote</i> <i>identify</i>
attributive adjective	attribute value of the noun that the attributive adjective modifies	[3, 43]	A large library has many sections. 'large' is the value of the attribute size of the class Library.
doing verb	operation	[3]	'submits' are doing verbs
having verb	aggregation	[3]	'has' and 'consists of' are having verbs
verb/verb phrase	association	[14, 60]	Two trains following each other. 'following' is the verb connecting two objects; therefore a reflexive association is identified for class Train.
property sentences	aggregation association	[43]	The university contains 10 departments.
universal quantifier (first entity) + unique existential quantifier (second entity)	many-to-one association	[43]	A complex aircraft uses the radar.
singular (first entity) + quantified by the definite article (second entity)	one-to-many association	[43]	The student passed all exams.
singular (first entity) + singular (second entity)	one-to-one association	[43]	The student passed the exam.
specific number	multiplicity	[43]	The student passed 3 exams.

Table 5 Summary of transformation rules (part 2)

Requirements concepts	Transformation rule		Rel. work	Example
	OO concepts	Constraint		
being verb	inheritance/generalization		[3]	'is a kind of' is a being verb
modal verb	constraints		[3]	'must be' is a modal verb.
verb/verb phrase	behavior	verb, predicate contains <i>has a capability to</i> <i>can</i> <i>able to</i>	[3, 106]	The student has a capability to learn. to learn is the behavior of the student.
direct object	message	sentence structure like <i>subject-direct object-indirect object</i>	[31]	The clerk sends the status of the load_bay to the system.
transitive verb	message	sentence structure like <i>subject-transitive verb-object</i>	[31]	The attendant enables the pump.
basic flow, alternative flow of use cases	sequence diagram		[49]	

2.7 Open issues and suggestions

As we have discussed in Section 2.6, a desirable approach, involving acceptable user effort in documenting requirements, should be able to (semi) automatically and efficiently generate a complete (i.e., including both static and dynamic aspects of a system) and consistent analysis model. Since none of the existing approaches achieves this, based on the systematic review results, the goal of this section is three-fold. We want to identify recurring issues in the research and reporting of the primary studies we reviewed, highlight open issues in existing solutions, and identify useful avenues of research.

2.7.1 Approach configuration

In this section, we first discuss the open issues identified for each aspect in an approach configuration. Then we recommend an approach configuration which, with due research, should be able to provide a solution to automatically and efficiently generate complete analysis models, based on acceptable user effort in documenting requirements.

2.7.1.1 Requirements configuration

A desirable requirements configuration should be able to effectively facilitate transformation from requirements to analysis models, while minimizing user effort in documenting requirements. However tradeoffs exist between the difficulty of following a requirements configuration and the extent to which it facilitates transformation, especially automated transformation:

1) Some approaches require additional DSI (Section 2.4.2.1) as requirement supplements; however, these approaches rely on users to manually provide DSI so that a great deal of user effort is required. We believe that demanding a textual glossary as a requirements supplement could be practical and requiring a domain model or definition could lower the representation gap between requirements and analysis models. However, it would be desirable to generate such a domain model automatically from requirements, at least an

initial version to be refined, rather than asking users to provide it. Furthermore, if the modeling of DSI is required, this should be well supported by tools.

2) Other approaches do not use any representation to structure their requirements (i.e., pure textual specifications); however if requirements are structured (e.g., using use case templates) one can expect that transformations be greatly facilitated. Almost half of the approaches require anyway that their requirements be documented using some form of use case template. Besides, use case modeling is commonly applied in practice. Therefore, we suggest having use cases, using appropriate templates, as the means of documenting requirements to facilitate automated transformations. Whether a use case template is easy to apply and whether it is able to effectively facilitate automated transformations should be experimentally investigated (Chapter 12).

3) Restricted NL is sometimes used for documenting requirements; however the rationale for restriction rules is often not clearly justified. Our summary table regarding restriction rules (Table 3), provides the rationale of each rule (Column 4 of the table), but we had to devise them by carefully examining each primary study since this information was in most cases not provided. It is important to know why a particular restriction rule is applied because further research may relax it by, for example, using new or improved NL analysis techniques. It is also paramount to know whether a set of restriction rules is easy to apply and whether its application can lead to a higher quality of automatically derived analysis models (Chapter 12). Again, experimental evaluations are required to further investigate this issue.

In summary, we believe that (i) it is desirable not to require additional DSI, though it may be practical to demand a textual glossary, (ii) use cases should be supported as they are most frequently used for requirements representation, and (iii) restricted NL might be used for documenting requirements so that automated transformation can be facilitated. Using our tuple representation—(DSI information, requirement representation, NL requirement)—we therefore, recommend that the following set of requirements configurations be considered in future work: (*None, Use cases, No*), (*None, Use cases,*

Yes), (*Glossary, Use cases, Yes*), and (*Glossary, Use cases, No*). We also recommend that experimental evaluations be performed to evaluate a requirements configuration method in terms of its applicability and effectiveness at automatically deriving analysis models.

It is worth noticing that requirements are not stand-alone artifacts; goals, assumptions, standards, and risks are all part of a complete requirements document. However, in the context of MDA, in order to generate an object-oriented analysis model, object-oriented analysis and design methodologies mostly use functional requirements as input for this specific transformation. Higher-level requirements artifacts such as goals, assumptions, standards and risks usually form a basis and justification for deriving detailed functional requirements (represented as use cases), which can then be further used to derive analysis models. Most of the primary studies identified by our systematic review take functional requirements (represented as use cases) as input to generate analysis models (Table 2) but the rest only use unstructured text as input (e.g., [68] and [48]), therefore not specifying the type of requirements they use in input.

2.7.1.2 Analysis model

As we have observed in Section 2.6.2, UML diagrams are most frequently used in the reviewed approaches to represent analysis models. This conforms to the MDA [57] transformation concept, which requires the source (e.g., PIM) and target (e.g., PSM) of a transformation to be represented as UML models. UML is a standardized language, is widely supported by a growing body of tools (e.g., [46]), open source plugins (e.g., [29]), and has been specialized for many domains.

If use case models, including use case diagrams and use case specifications, are used to structure and document requirements and UML models are used as the representation of the analysis model, a relationship can be clearly established between the use case models and parts of the analysis models. In particular, since use case descriptions describe interactions of the system and actors along the time line, they can be transformed into messages in sequence diagrams, an important component of behavioral modeling in

analysis models. With an appropriate use case template, it is expected that conditions and branches in use case specifications can be automatically captured and transformed into *CombinedFragments* [77] in sequence diagrams. In addition, extend and include relationships in use case specifications can be transformed into *InteractionUse* [77] of sequence diagrams.

UML models can model not only the structural aspect of a system (e.g., class diagrams), but also the behavioral aspects (e.g., sequence and activity diagrams). Though this is to some extent dependent on the modeling method used, consistency between the structural and behavioral aspects can be easily achieved in the context of UML since when transformations are performed, one single UML model is created, queried, and maintained during the transformations; different diagrams are just different, overlapping views of the same underlying model.

Therefore, for the above practical and technical reasons, we suggest using UML models as the representation of analysis models.

A methodological open issue we identified in this review is that many of the approaches cannot generate a complete analysis model (i.e., both structural and behavioral aspects). Additionally the correctness of their generated analysis models is not evaluated. The quality of an automatically generated analysis model should be evaluated by, for example, comparing it with existing expert solutions to see how close the automated analysis model is to these expert solutions.

2.7.1.3 Automation

The level of automation is one of the important characteristics of transformations. Automated transformations are always desired; however when a certain amount of manual intervention is indispensable for documenting requirements, performing transformations, or establishing traceability links, it should be explicitly described and its expected effort should be evaluated. For automated approaches, transformation

algorithms should be clearly specified, and this is a requirement which is not always met in the approaches of this review.

2.7.1.4 Efficiency

Our evaluation results show that most of our reviewed approaches need complicated requirements pre-processing, contain two or more transformation steps, and/or user intervention is required in many places. In terms of requirements pre-processing techniques, some approaches require significant user effort to manually pre-process requirements, for example, manual categorization of requirements (Section 2.4.2.3). We suggest that only automatable NL processing techniques should be used. Since it is paramount to automate transformations, user's involvement should be minimized. Additionally, the more intermediate models, the more difficult the validation and verification of the approach; the more intermediate models, the higher the chances of losing information during the transformation from requirements to analysis models (because of multiple transformations). However the complexity of transformations and amount of information to manipulate suggest that not relying on an intermediate model might be difficult to achieve. Indeed, most of our reviewed approaches have one intermediate model (two transformations). Last, we will argue in Section 2.7.3 that one intermediate is necessary. Therefore we suggest that a maximum of one intermediate model be required in an approach.

According to above discussion, we recommend the following set of approach configurations:

Automatically transform use case models with or without restricted NL and/or glossaries to complete (i.e., including both static and dynamic aspects), correct and consistent UML models using one intermediate model and fully automatable requirements pre-processing techniques (e.g., lexical analysis and syntactic parsing).

2.7.2 Intermediate model

Some approaches use intermediate models as bridges for transformation between requirements and analysis models. The main reason is that requirements are usually text-based, and automated transformations (to fully integrate requirements into model-driven approaches) cannot be easily supported with unrestricted, unstructured requirements representations such as pure text. The reason for using one specific type of intermediate models should be explicitly justified in the research literature and the following considerations should be taken into account when intermediate models are selected:

- The representation of the source and target models since they drive the selection of intermediate models (if any) as well as transformation rules.
- Whether the intermediate model(s) can be easily integrated into existing tool support.
- If user interventions are required during transformations, it is important that the intermediate model be easy to understand by users.
- Whether the intermediate model is general enough to be used for multiple purposes, such as generating not only class diagrams, but also sequence diagrams, activity diagrams, and state machines. The intermediate model KCMP [33] is one such example.
- Whether it can be used independently of different NL processing techniques.
- Whether it is suitable to support traceability analysis.

The above items are usually not carefully discussed in most primary studies. As a result, the proposed technologies are often difficult to assess.

2.7.3 Transformations

In this section, we discuss open issues and our recommendations on transformations from the following aspects: transformation approaches (Section 2.7.3.1), traceability support (Section 2.7.3.2), transformation algorithm (Section 2.7.3.3), and the transformation quality characteristics (Section 2.7.3.4) such as efficiency and scalability.

2.7.3.1 Approach

As discussed in Section 2.4.2.5, four types of transformation approaches are applied in the primary studies we have reviewed. Selecting which transformation approach to apply is closely related to the representations of source and target models, the complexity and scalability of transformations, and the extent of automation which is targeted.

A classification of transformation approaches is reported in [23, 24], along with a high-level discussion on pros and cons of each type of transformation approaches. For rule-based and pattern-based transformation approaches, as indicated in [23, 24], transformation rules¹² should clearly specify, for example, their application domains, parameters, application constraints, and directions. None of the primary studies of this review clearly specify their transformation rules according to these aspects.

There exist techniques in academia and commercial tools that can facilitate the specification and execution of transformation rules. The Atlas Transformation Language (ATL) [8, 53] is one such model to model transformation technique, developed on top of the Eclipse platform [28], to facilitate the specification, structuring (by packaging rules into modules), and execution of transformation rules. Besides, it provides both declarative and imperative constructs to define transformation rules. However, during the execution of an ATL transformation, its target model cannot be navigated. This often results in complex transformation rules since results from previously executed rules cannot be used as inputs of other rules. Kermet [55] is an imperative metamodeling

¹² In [23, 24], patterns are considered as one type of transformation rules.

language, also built on top of the Eclipse platform, which can facilitate the manipulation of both source and target model elements. Kermeta also supports packages, inheritance, classes, and operations so that transformation rules can be well organized. In addition, another interesting characteristic is design-by-contract for rules: operations implementing rules support pre and post conditions and classes use invariants. There are other academic and commercial tools and languages which can support model to model transformations, such as the IBM Model Transformation Framework (MTF) [47] and the Query/View/Transformation (QVT) standard [74]. A quite exhaustive list of such tools and languages can be found in [24]. We suggest utilizing an existing transformation framework to support transformation from requirements to an analysis model. However, requirements are usually textual, not models. Therefore, we suggest that requirements are transformed into an intermediate model, which can then be further transformed into an analysis model by applying one of the model-to-model transformation techniques.

Another open issue we identified in this review is that many of the approaches do not address the extent to which their generated analysis models are correct and precise enough. One possible evaluation method could be experimentally comparing the analysis model generated by the transformation approaches with the one manually developed by software developers. Research (e.g., [91]) has also been conducted to systematically test and thus validate transformation approaches themselves to ensure that they have the desired behavior. If possible, these approaches should be applied in our context.

2.7.3.2 Traceability support

Most of the approaches do not address traceability. This is perhaps because in order to support traceability, a mechanism should be proposed to establish and maintain explicit traceability links between the source and target of each transformation. In cases where multiple transformation steps are involved, traceability links should also be derived for requirements and analysis models from the established traceability links during each transformation step. A traceability link should at least contain references to the source and target elements connected by the link, and should preferably indicate which

transformation rule(s) are applied to trigger the creation of the link. Another interesting aspect, which is not addressed in any of the approaches, is that transformation rules may rely on the results of other transformation rules, more specifically transformation rules may rely on traceability links established by other transformation rules. One advantage is that transformation rules can thus be simplified. For example, instead of conducting analyses already performed by other rules, we can simply use traceability links. For example, suppose that a class has been generated from a requirement construct (e.g., a noun) and a traceability link has been established accordingly between the generated class and the requirement construct. If a new transformation rule identifies that this noun (requirement construct) is qualified by an adjective (another requirement construct), then the established traceability link can be used to create an attribute in the generated class. This way, the output (traceability link) of the first transformation rule is an input to the second. This mechanism is very useful for transformation approaches that need to query previously generated target elements and trace back to their corresponding source elements through the traceability links previously established.

2.7.3.3 Algorithm

Not all approaches do provide transformation algorithms. And when they do, they often do not describe their algorithms at a proper level of details that is amenable to an implementation. To facilitate automated transformation, an algorithm should be clearly specified, for example to describe how and when to apply transformation rules. A transformation algorithm should specify the sequence of applying transformation rules, when there are sequential constraints among them. For example, to generate sequence diagrams, one must identify objects before identifying messages exchanged between these objects. A transformation algorithm should also specify how to verify conditions triggering transformation rules. A well-designed algorithm should be easily modifiable when additional transformation rules are added or existing rules are modified. It is possible not to rely on transformation rules (i.e., a transformation is fully described in an algorithm); however, this strategy just works for very simple transformations, which is rarely the case. In cases with a large number of rules, the logic of the algorithm will

become very complex and modifications increasingly more difficult. We suggest clearly separating transformation rules from transformation algorithms applying them.

2.7.3.4 Quality characteristics

Ideally, we also expect transformation approaches to address quality characteristics such as efficiency, scalability, extensibility, and interoperability. Fine-grained transformation from requirements to an analysis model could be very complex; therefore efficiency and scalability of transformation approaches could become an issue for large software systems. Large-scale case studies are required to evaluate these two quality characteristics. In addition, we also suggest using a minimum number of intermediate models since additional intermediate models unavoidably make transformation approaches less efficient: the more intermediate models, the more difficult the validation and verification of the approach and the higher the chances of losing information during transformations. In terms of extensibility, it is important to be able to add new transformation rules easily and modify transformation algorithms without too many side effects. It is also desirable that a proposed transformation approach be easily integrated with other approaches or tools, used within a software engineering process such as approaches transforming analysis models into design models, and code generation.

2.8 Conclusion

In the context of model-driven development, the early step of transforming requirements into an analysis model is a crucial but difficult step. Although mostly performed manually, there have been attempts to automate this software development step. However, despite a significant amount of research, we still do not have a practical, workable automated solution. To gain a precise and structured understanding of the state of the art and identify directions for future research, this paper provides a systematic review of existing work on automating this step. This review systematically selected, investigated, and compared 16 approaches for transforming requirements into an analysis model.

In order to facilitate the synthesis and comparison of the approaches in a systematic manner, a conceptual framework was designed to provide common concepts and terminology for the comparison and evaluation of transformation technologies. This framework also includes a description of the general steps of transforming requirements into an analysis model while establishing traceability links. A set of evaluation criteria, which are derived from the conceptual framework, is proposed to assess each approach in a precise and structured manner. These evaluation criteria can be adapted to evaluate future research works on the same topic.

Based on the systematic review results, we observed that no existing approach, (i) requires acceptable user effort to document requirements, (ii) is efficient enough (e.g., one or two transformation steps), (iii) is able to (semi-)automatically generate a complete (i.e., static and dynamic aspects), consistent analysis model, which is expected to model both the structure and behavior of the system at a logical level of abstraction, e.g., UML models that at least contain consistent class and interaction diagrams. However, by carefully analyzing and evaluating each aspect of the reviewed approaches, we can make recommendations for future work and a desirable approach can be outlined. A desirable approach is one that can automatically and efficiently transform a use case model using reasonable restrictions to natural language, with or without domain specific information provided in a glossary, into a complete, correct and consistent UML model comprising both structural and behavioral aspects using one intermediate model and fully automatable requirements pre-processing techniques.

Additionally, our review results show that four types of transformation approaches are applied in the reviewed approaches and selecting which transformation approaches to apply is closely related to multiple factors such as the representation of requirements and analysis models. Existing model to model transformation techniques (e.g., ATL [8] and Kermeta [55]) can be adopted to implement a requirements-to-analysis model transformation approach. Transformation rules and algorithms should be clearly structured and specified. We also summarize and classify transformation rules applied in the reviewed works for future research reference. Our review results also show that most

of the approaches do not address traceability. We suggest that a traceability mechanism should be proposed to create and maintain traceability links between requirements elements and analysis model elements. In cases where intermediate models are used, traceability links should also be derived all the way from requirements, through the intermediate models, to the analysis model. Last, we also suggest that research on transformation approaches address, in part through empirical studies, their quality characteristics such as usability, efficiency, scalability, extensibility, and interoperability.

2.9 Other related work

There exist several works that were recently published and therefore were not included in our systematic literature review. Śmiałek et al [93] propose an approach to automatically transform use cases in restricted NL into sequence diagrams. Several simple transformation rules are discussed in the paper. The approach requires a user to manually parse NL sentences by indicating their subject and predicate. In addition, the user has to manually construct a vocabulary that specifies all the subjects, verbs, objects of the sentences. We can see that a substantial user effort is required to write use cases and build the vocabulary. A tool has been developed and used in four industry areas to validate the approach. Users' opinion on the tool was collected through questionnaire and overall the users are positive to the tool. Gutiérrez et al [42] propose an approach to generate activity diagrams from use cases. The objective is to eventually derive test cases from automatically generated activity diagrams. Fortuna et al [37] propose an approach to derive domain models (class diagrams) from formalized use cases, assuming a dictionary specifying the objects involved in use cases exists. Substantial user effort is required since most of the class diagram output is already part of the input. Ravenflow [85], Visual Paradigm for UML [104], and CaseComplete [17] are three commercial tools which can generate activity diagrams from textual requirements. Both have a different objective than ours: helping analysts to elicit, specify and validate business-level requirements by visualizing textual requirements as activity diagrams. None of these more recent approaches are able to generate analysis class and sequence diagrams from use case models.

As discussed earlier, some approaches have been proposed to manually, semi-automatically or automatically transform unrestricted or restricted requirements into formal specifications (e.g., [66, 86, 103]) but their purpose is requirements analysis instead of model generation. The similarity between this school of research and our work is that Natural Language Processing (NLP) techniques are applied to extract required information from textual requirements. However, the objectives and outputs are different. Non-automated methods (e.g., [18]) have been suggested to transform NL requirements (for instance under the form of use cases) into models such as Use Case Maps [54] and statecharts [87].

CHAPTER 3 RUCM

Use case modeling, including use case diagrams and UCSs, is commonly applied to structure and document requirements. UCSs are usually structured, unrestricted textual documents complying with a certain use case template. However, because UCSs remain essentially textual, ambiguity is inevitably introduced. In this thesis, we propose a use case modeling approach, called Restricted Use Case Modeling (RUCM), which is composed of a use case template that merges several aspects of existing templates (Section 3.2) and a set of well-defined restrictions to the use of plain language for documenting UCSs (Section 3.3). The goal is two-fold: (1) restrict the way users can document use case specifications in order to reduce ambiguity and (2) facilitate automated analysis in order to provide tool support to derive initial analysis models, which in UML are typically composed of class diagrams, interaction diagrams, and possibly other types of diagrams and constraints. The related work is reported in Section 3.1. An example of applying our restriction rules and the template is provided in Section 3.4. The controlled experiment conducted to evaluate RUCM is discussed in Chapter 12.

3.1 Related work

Two streams of research relate to our work: use case templates (Section 3.1.1) and restriction rules (Section 3.1.2).

3.1.1 Use case template

The concept of use case has been first introduced by Ivar Jacobson in 1986 to capture functional requirements [50]. Since then, use cases have been widely accepted and use case modeling techniques have evolved and matured. The concept of use case is part of the UML (though UML does not support use case specification), which provides a use case diagram to specify relations between use cases and between use cases and actors. As the importance of use cases (requirements) is increasingly recognized, use case modeling is more than a requirements specification technique; it drives the whole software

development process as most activities (e.g., analysis, design, and test) start from use cases. This is referred to as use case-driven software development [52].

It is a common practice to follow a use case template to structure UCSs, thereby helping their writing, reading and reviewing. Various use case templates (e.g., [21, 52, 58, 59, 61]) have been suggested in the literature to satisfy different application contexts and purposes. They share common fields such as: *use case name*, brief overall *description*, *precondition*, *postcondition*, *basic flow*, and *alternative flows*. The template suggested in [21] introduces additional fields such as *scope*, *level* (level of abstraction) and *stakeholders and interests*. The Rational Unified Process (RUP) [58] suggests a template with six fields: *use case name*, *flow of events* (basic flow, alternative flows), *special requirements* (e.g., performance and security requirements), *preconditions*, *postconditions*, and *extension points*. Kulak and Guiney [59] suggest a template that includes an *exception paths* field: exception paths are distinguished from alternative flows since they are paths taken when errors occur. Both Cockburn and Larman [21, 61] mention that use cases can be written at different levels of details: brief, casual, and fully dressed. The templates they propose are similar and composed of most of the fields that have been proposed in the literature and that we mentioned previously. In addition, a template for describing actors is proposed in [7], which consists of three sections: *actor name*, *description*, and *examples*.

In addition to capturing requirements, use cases can also facilitate the automated derivation of an initial analysis model – one of our goals. The systematic literature review (Chapter 2) we conducted to examine techniques that transform textual requirements into analysis models revealed that six approaches require use cases. The approach proposed in [63] relies on the RUP use case template [58]. The use case template proposed in [95] contains eight fields: title (i.e., use case name), primary actor, participants (i.e., secondary actors), goal (i.e., brief description), precondition, postcondition, steps (i.e., basic flow steps), and alternatives. An enriched use case template is proposed in [49], which is composed of three sections: use case summary, basic flows, and alternative flows. The use case summary section is further divided into four subsections: use case name, actors,

cross-reference, and an overview of the use case purpose (may be used to describe non-functional requirements). The cross-reference section is used to link the use case to high-level requirements. A three-column structure (i.e., general, actor/system communication, and system response) is introduced in this use case template to structure the steps of flow of events. The general column describes general activities that are not supposed to be implemented by the system but can help users better understand the use case. Steps in the actor/system communication column specify actions performed by actors when interacting with the system. The system response column represents reactions of the system, including changes of state.

The use case template we propose in this thesis (Section 3.2) contains fields similar to those encountered in conventional use case templates but with a few variations on the structure of the flow of events. The motivation is to support the automated generation of analysis models and to further reduce ambiguity. Our objective is to propose a new use case template that not only complies as much as possible with conventional use case templates but also facilitates the process of deriving analysis models. Therefore, we made the following decisions: (1) We included fields commonly encountered in most templates: use case name, brief description, primary actor, secondary actor, precondition, postcondition, basic and alternative flows. (2) Some of the fields (e.g., *scope*, *level*, and *special requirements*) proposed in the literature to capture requirements were excluded since they do not help deriving analysis models. (They could easily be added though.) (3) We excluded the fields that, on the one hand may increase the precision of UCSs but, on the other hand require that the designer provide much more information than what is actually needed for our purpose. In other words, we believe that the additional precision does not warrant the additional cost, and that these fields do not bring clear advantages with respect to our objectives. For example, the semantics of the three-column steps modeling style proposed in [49] (discussed above) can actually be automatically derived from UCSs by grammatically analyzing each sentence; therefore we made a design decision not to include this style into our use case template. (4) Six interactions types (five from [21], one we newly propose in this work) are suggested to describe action

steps of flow of events. (5) Differing from most of existing use case templates that suggest having one postcondition for one use case, our template enforces that each flow of events (both basic flow and alternative flows) of a UCS contains its own postcondition: the postcondition of the use case is simply the disjunction of all those postconditions.

3.1.2 Restriction rules

In Section 2.6.5, we summarize and classify the restriction rules (also called writing guidelines) applied in [92, 95, 98, 106], which propose transformations from requirements to analysis models. In this thesis, we propose a total of 26 restriction rules on the use of NL to document UCSs that complies with our use case template. Some of these restrictions are the results of the systematic review (Section 2.6.5) we conducted (see further details in Section 3.3); others are heuristics suggested in the literature on writing use cases (e.g., [4, 11, 21]); last some of them are derived from our experience in writing UCSs when attempting to reduce ambiguity and facilitating automated transition to analysis models. None of the related work we looked at relies on a set of rules as complete as the one we suggest.

Existing guidelines are recommended, based on practitioners' experience in writing UCSs, to reduce ambiguities of UCSs or to facilitate the process of deriving analysis models from them. We reused some of them, excluded others, recommended new ones, and classified all the rules. For example, we excluded the rules that constrain grammatical sentence structures, such as only allowing certain types of structures such as subject-verb-object (e.g., [4, 22]), because these structures can be automatically obtained by grammatically analyzing each sentence using a NL parser. We also excluded the rules that put excessive constraints on wording. For example, one such rule suggests using "is a kind of", "is specification of" or "is generalization of" to describe inheritance between the subject and object of a sentence.

Additionally, we explicitly describe why each of our restriction rules is needed either to reduce ambiguities or facilitate the process of deriving analysis models, a crucial piece of information that is often omitted in the literature. We also indicate how and where to apply (Section 3.3) each of our restriction rules, another piece of information often left out by most papers on the topic. Several rules we newly propose in this work are based on our experience with several NL parsers (e.g., [101]) and are proposed because sentences with certain structures cannot be correctly parsed. These rules can also help reduce ambiguity of UCSs and therefore help to manually derive analysis models from them. Furthermore, as opposed to many related works, our restriction rules are integrated with our use case template together as a comprehensive solution for use case modeling: several of our restriction rules refer to some of the features of our use case template (Section 3.3).

3.2 Use case template

Our use case template has eleven first-level fields (first column of Table 6). The last four fields are decomposed into second-level fields (second column of the last four rows). The last column of each row explains the corresponding field(s). There is no need to further discuss the first seven fields since they are straightforward and commonly encountered in many templates. Below we focus the discussion on the *Basic Flow* field and the three types of *Alternative Flows*: specific, global, and bounded alternative flows.

A *basic flow* describes a main successful path. It often does not include any condition or branching [61]. It is recommended to describe separately the conditions and branching in alternative flows. A basic flow is composed of a sequence of steps and a postcondition. Each UCS can only have one basic flow. The action steps can be one of the following five interactions, which are reused from [21] except for the fifth:

1. Primary actor→system: the primary actor sends a request and data to the system.
2. System→system: the system validates a request and data.

3. System→system: the system alters its internal state (e.g., recording or modifying something).
4. System→primary actor: the system replies to the primary actor with a result.
5. System→secondary actor: the system sends requests to a secondary actor.

All steps are numbered sequentially. This implies that each step is completed before the next one is started. If there is a need to express conditions, iterations, or concurrency, then specific keywords, specified as restriction rules in Section 3.3, should be applied.

Alternative flows describe all the other scenarios or branches, both success and failure. An alternative flow always depends on a condition occurring in a specific step in a flow of reference, referred to as *reference flow*, and that reference flow is either the basic flow or an alternative flow itself. The branching condition is specified in the reference flow by following restriction rules (R20 and R22—Section 3.3). We refer to steps specifying such conditions as *condition steps* and the other steps as *action steps*. Similarly to the basic flow, an alternative flow is composed of a sequence of numbered steps. We classify alternative flows into three types: specific, global, and bounded alternative flows. This classification is adapted from [11]. A *specific alternative flow* is an alternative flow that refers to a specific step in the reference flow. A *bounded alternative flow* is an alternative flow that refers to more than one step in the reference flow—consecutive steps or not. A *global alternative flow* (called *general alternative flow* in [11]) is an alternative flow that refers to any step in the reference flow. Distinguishing different types of alternative flows makes interactions between the reference flow and its alternative flows much clearer. For specific and bounded alternative flows, a RFS (Reference flow Step) section, specified as rule R19 (Section 3.3), is used to specify one or more (reference flow) step numbers. Whether and where the flow of an alternative flow merges back to the reference flow or terminates the use case must be specified as the last step of the alternative flow. Similarly to the branching condition, merging and termination are specified by following restriction rules (R24 and R25—Section 3.3). By doing so, we can avoid potential ambiguity in

UCSs, caused by unclear specification of interactions between the basic flow and its corresponding alternative flows. Each alternative flow must have a postcondition (enforced by restriction rule R26—Section 3.3).

It is usual to provide a postcondition describing a constraint that must be true when a use case terminates. If the use case contains alternative flows, then the postcondition of the use case should describe not only what must be true when the basic flow terminates but also what must be true when each alternative flow terminates. The branching condition to each alternative flow is then necessarily part of the postcondition (to distinguish the different results). In such a case, the postcondition can become complex and the branching condition for each alternative flow is redundantly described (both in the steps of flows and the postcondition), which therefore increases the risk of ambiguity in UCSs. Our template enforces that each flow (both basic flow and alternative flows) of a UCS contains its own postcondition and therefore avoids such ambiguity.

Table 6 Use case template

Use Case Name	The name of the use case. It usually starts with a verb.	
Brief Description	Summarizes the use case in a short paragraph.	
Precondition	What should be true before the use case is executed.	
Primary Actor	The actor which initiates the use case.	
Secondary Actors	Other actors the system relies on to accomplish the services of the use case.	
Dependency	Include and extend relationships to other use cases.	
Generalization	Generalization relationships to other use cases.	
Basic Flow	Specifies the main successful path, also called “happy path”.	
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the basic flow executes.
Specific Alternative Flows	Applies to one specific step of the basic flow.	
	RFS	A reference flow step number where flow branches from.
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.
Global Alternative Flows	Applies to all the steps of the basic flow.	
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.
Bounded Alternative Flows	Applies to more than one step of the basic flow, but not all of them.	
	RFS	A list of reference flow steps where flow branches from.
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.

3.3 Restriction rules

The restriction rules are classified into two groups: restrictions on the use of natural language, and restrictions enforcing the use of specific keywords for specifying control structures. The first group of restrictions is further divided into two categories according to their location of application (see below). Each restriction rule is assigned a unique number.

Seven restriction rules (R1-R7) constrain the use of natural language (Table 7): the table explains why they are needed to reduce ambiguity. Notice that these rules apply only to action steps; they do not apply to condition steps or preconditions or postconditions. The other nine restriction rules (Table 8) apply to all sentences in a UCS: action steps, condition steps, preconditions, postconditions, and sentences in the brief description. Rules R8-R10 and R16 are to reduce ambiguity of UCSs; the remaining rules specifically facilitate automated generation of analysis models, though they can also help reduce ambiguity. These two sets of restrictions are thought to be good practice for writing clear and concise UCSs (e.g., [11, 21, 90]) except for R13 and R15. We include these two rules because we perceived that negative adverbs, negative adjectives, and participle phrases are very difficult to parse for natural language parsers. R9 requires using words consistently to document UCSs. A common approach to do so is to use a domain model and glossary (e.g., [14, 61]) as a basis to write UCSs.

Table 7 Restriction rules (R1-R7)

#	Description	Explanation
R1	The subject of a sentence in basic and alternative flows should be the system or an actor.	Enforce describing flows of events correctly. These rules conform to our use case template (the five interactions).
R2	Describe the flow of events sequentially.	
R3	Actor-to-actor interactions are not allowed.	
R4	Describe one action per sentence. (Avoid compound predicates.)	Otherwise it is hard to decide the sequence of multiple actions in a sentence.
R5	Use present tense only.	Enforce describing what the system does, rather than what it will do or what it has done.
R6	Use active voice rather than passive voice.	Enforce explicitly showing the subject and/or object(s) of a sentence.
R7	Clearly describe the interaction between the system and actors without omitting its sender and receiver.	

Table 8 Restriction rules (R8-R16)

#	Description	Explanation
R8	Use declarative sentence only. "Is the system idle?" is a non-declarative sentence.	Commonly required for writing UCSs.
R9	Use words in a consistent way.	Keep one term to describe one thing.
R10	Don't use modal verbs (e.g., <i>might</i>)	Modal verbs and adverbs usually indicate uncertainty; Instead, metrics should be used if possible.
R11	Avoid adverbs (e.g., <i>very</i>).	
R12	Use simple sentences only. A simple sentence must contain only one subject and one predicate.	
R13	Don't use negative adverb and adjective (e.g., <i>hardly, never</i>), but it is allowed to use <i>not</i> or <i>no</i> .	Facilitate automated natural language parsing and reduce ambiguity.
R14	Don't use pronouns (e.g. <i>he, this</i>)	
R15	Don't use participle phrases as adverbial modifier. For example, the italic-font part of the sentence "ATM is idle, <i>displaying a Welcome message</i> ", is a participle phrase.	
R16	Use "the system" to refer to the system under design consistently.	

The remaining ten restriction rules (Table 9) specify control structures, except R26 that specifies that each basic flow and alternative flow should have its own postcondition. R17 and R18 specify keywords to describe use case dependencies *include* and *extend*. R19 specifies keyword RFS, which is used in a specific (or bounded) alternative flow to refer to a step number (or a set of step numbers) of a reference flow that this alternative flow branches from.

Rules R20-R23 specify the keywords used to specify conditional logic sentences (IF-THEN-ELSE-ELSEIF-ENDIF), concurrency sentences (MEANWHILE), condition checking sentences (VALIDATES THAT), and iteration sentences (DO-UNTIL), respectively. The keyword IF-THEN-ELSE-ELSEIF-ENDIF can be used in three different ways (these are specified as a grammar): 1) IF-THEN-ENDIF (appears in one flow only), 2) IF-THEN-ELSE-ENDIF (everything is in one flow, or IF-THEN in one flow and ELSE in an alternative flow), and 3) IF-THEN-ELSEIF-THEN-ELSE-ENDIF (everything is in one flow, or IF-THEN in a flow and ELSEIF-THEN-ELSE-ENDIF in an alternative flow). Keyword VALIDATES THAT (R22) means that the condition is evaluated by the system and must be true to proceed to the next step. This rule also requires an alternative flow describing what happens when the validation fails (the

condition does not hold). Rules R20 and R22 are two complex rules, when compared with the others of the same rule set, because both of them require that UCS designers look at multiple steps in two different flows: the basic flow and an alternative flow.

R24 and R25 specify keywords ABORT and RESUME STEP to describe an exceptional exit action and when an alternative flow goes back to its corresponding basic flow, respectively. These two rules also specify that an alternative flow ends either with ABORT or RESUME STEP, which means that the last step of the alternative flow should clearly specify whether the flow returns back to the basic flow and where (using keywords RESUME STEP followed by a returning step number) or terminates (using keyword ABORT).

R17-R21 and R23 have been proposed in the literature and we reused them with some variation. We add R22, R24 and R25 for the purpose of making the whole set of restrictions as complete as possible so that flows of events and interactions between the basic flow and the alternatives can be clearly and concisely specified. Applying this set of rules facilitates automated NL processing (e.g., correctly parse sentences with our specified keywords) and generating of analysis models, especially sequence diagrams which also helps reducing ambiguity of UCSs (Chapter 7 and Chapter 8). The detailed description of all the 26 restriction rules is provided in Appendix A.

Table 9 Restriction rules (R17-R26)

#	Description	#	Description
R17	INCLUDE USE CASE	R22	VALIDATE THAT
R18	EXTENDED BY USE CASE	R23	DO-UNTIL
R19	RFS	R24	ABORT
R20	IF-THEN-ELSE-ELSEIF-ENDIF	R25	RESUME STEP
R21	MEANWHILE	R26	Each basic flow and alternative flow should have its own postconditions.

3.4 Example

An example of use case descriptions documented with our use case template and restriction rules are presented in Table 10. The original design of the use case

descriptions is from [39]. We rewrote them by applying RUCM. As show in Table 10, the use case `Withdraw Funds` contains one basic flow, one specific alternative flow, one bounded alternative flow, and one global alternative flow. The specific and bounded alternative flows correspond to four basic flow steps containing the keyword `VALIDATES THAT`. Notice that this is just one possible specification of the UCS applying our template and restrictions; it is possible to have different but equivalent specifications: for example, using the keyword `IF-THEN-ELSE-ELSEIF-ENDIF` instead of `VALIDATES THAT`.

Table 10 Use case Withdraw Funds

Use Case Name	Withdraw Funds	
Brief Description	ATM customer withdraws a specific amount of funds from a valid bank account.	
Precondition	The system is idle. The system is displaying a Welcome message.	
Primary Actor	ATM customer	
Secondary Actors	None	
Dependency	INCLUDE USE CASE Validate PIN.	
Generalization	None	
Basic Flow	Steps	
	1	INCLUDE USE CASE Validate PIN.
	2	ATM customer selects Withdrawal.
	3	ATM customer enters the withdrawal amount.
	4	ATM customer selects the account number.
	5	The system VALIDATES THAT the account number is valid.
	6	The system VALIDATES THAT ATM customer has enough funds in the account.
	7	The system VALIDATES THAT the withdrawal amount does not exceed the daily limit of the account.
	8	The system VALIDATES THAT the ATM has enough funds.
	9	The system dispenses the cash amount.
	10	The system prints a receipt showing transaction number, transaction type, amount withdrawn, and account balance.
	11	The system ejects the ATM card.
	12	The system displays Welcome message.
	Postcondition	ATM customer funds have been withdrawn.
Specific Alternative Flows	BFS 5-7	
	1	The system displays an apology message MEANWHILE the system ejects the ATM card.
	2	The system shuts down.
	3	ABORT.
	Postcondition	ATM customer funds have not been withdrawn. The system is shut down.
Global Alternative Flows	IF ATM customer enters Cancel THEN	
	1	The system cancels the transaction MEANWHILE the system ejects the ATM card.
	2	RESUME STEP Basic Flow 12
	ENDIF	
	Postcondition	ATM customer funds have not been withdrawn. The system is idle. The system is displaying a Welcome message.
Bounded Alternative Flows	BFS 8	
	1	The system displays an apology message MEANWHILE the system ejects the ATM card.
	2	ABORT.
	Postcondition	ATM customer funds have not been withdrawn. The system is displaying an apology message.

CHAPTER 4 UCMETA

UCMeta is the intermediate model in aToucan, used to bridge the gap between textual UCMods and UML analysis models. As a result, the transformation is divided into two steps: the transformation from the textual UCMOD to the intermediate model (FormalizeUCM) and from the intermediate model to the analysis model (GenerateUML). We chose not to directly transform textual UCMods to UML analysis models because: 1) requirements are usually text-based and automated transformations cannot be easily supported with unrestricted, textual requirements representations such as UCMods, and 2) with the intermediate model, different types of diagrams can be generated such as class, sequence, activity diagrams without repeating the inevitable step of parsing the UCS NL sentences; thereby improving the overall efficiency of the transformations.

We investigated whether one of the existing approaches to formalize use case models (referred to as use case metamodels) could be the intermediate model of our method. The approaches we found [25, 70, 78, 92, 102] fail to satisfy our objectives for one or more of the following reasons: Significant NL information is not captured by the metamodel. No sentence-level concepts can be described with the metamodel. We therefore propose a use case metamodel, specified using Model Object Facility (MOF) [73], as the intermediate model of our method. The decision of using MOF to specify our use case metamodel is motivated by the following considerations: 1) MOF has been used to describe the most commonly used metamodels like UML and therefore it can be used to specify UCMeta, 2) UML is the target model of our transformation, thus facilitating the transition from our intermediate model (also based on MOF) to UML with a simplified tool implementation. This metamodel, denoted as UCMeta, captures all the necessary information required for the subsequent transformation (GenerateUML), including a use case diagram, a use case template, and NL information in textual UCSs. The NL information covered by UCMeta comes from four resources: English grammar (e.g.,

[40]), Linguistics (e.g., [13]), NLP books (e.g., [65]), and our experience on documenting UCSs. UCMeta also complies with the restrictions and use case template of RUCM.

UCMeta consists of 108 metaclasses and is expected to evolve over time. In the rest of the section, we describe its architecture and packages at a high level of abstraction. The dictionary of the metaclasses of UCMeta is provided in Appendix B for reference. The UCS in Table 10 is used as a running example. Notice that UCMeta could also be used, as a formal specification language, for other purposes such as requirements analysis, not the focus of this paper though.

4.1 Architecture

The architecture of UCMeta is hierarchical, containing three layers: Figure 9. The top layer is package `UCMeta`, which imports four middle-layer packages: `UML::UseCases`, `UCSTemplate`, `SentencePatterns`, and `SentenceSemantics`. Package `UCSTemplate` further imports package `SentenceStructure` (the third layer). UCMeta has been architected with the following design principles in mind:

a) *Modularity*

We group all related concepts into packages by following the principle of strong cohesion within a package and loose coupling between packages;

b) *Layering*

We decompose UCMeta into layers to form a hierarchical structure. For example, the NL concepts are separated from the concepts of use case template, which are separated from basic UML use case concepts. Changes to the middle layer impact the top layer, but not the bottom layer;

c) *Partitioning*

We group concepts in a same layer into peer packages. For example, the middle

layer of the architecture is partitioned into four packages.

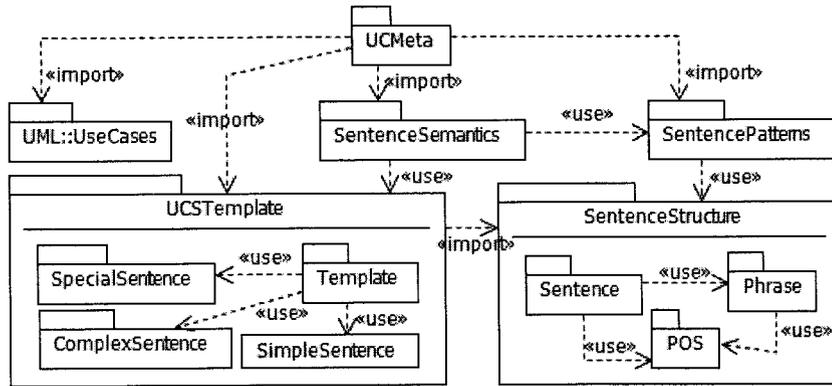


Figure 9 Architecture of UCMeta

4.2 Packages

Package UCMeta contains two metaclasses: UseCaseModel and UseCaseModelElement: UseCaseModel is composed of UseCaseModelElement and extends it, as shown in Figure 10. Metaclass UseCaseModelElement is used as the common superclass of all the metaclasses of UCMeta; therefore metaclass UseCaseModel is also a subclass of UseCaseModelElement.

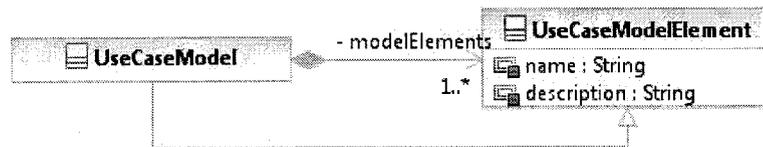


Figure 10 Package UCMeta of UCMeta

UML::UseCases is a package of UML 2 superstructure [77], which defines the key concepts used for modeling use cases: actors, use cases, and the system under design. As shown in Figure 11, ATM System is an instance of UseCaseModel composed of a set of use cases, one of which is Withdraw Funds (whose textual UCS is provided in Table 10 Use case Withdraw Funds).

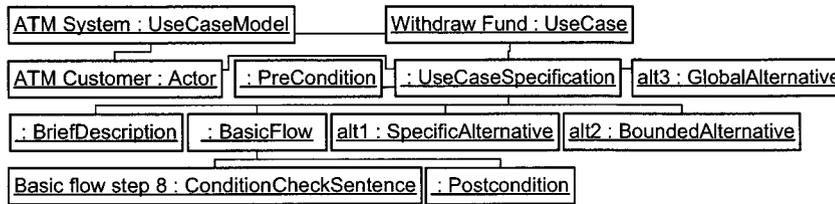


Figure 11 Example of package UCMeta

4.2.1 UCSTemplate

Package UCSTemplate models the concepts in the RUCM use case template (sub-package Template in Figure 9), as shown in Figure 12: those concepts model the structure that one can observe in Table 6. Metaclass UseCase of package UML::UseCases is extended by adding an association to metaclass UseCaseSpecification, which contains a BriefDescription, a Precondition, one or more FlowOfEvents, a primary actor (role name primaryActor), and zero-to-many secondary actors (role name secondaryActors). BriefDescription, PreCondition, FlowOfEvents, and PostCondition are all composed of Sentences. There are two types of flow of events: BasicFlow and AlternativeFlow. Each use case must have one single BasicFlow and zero-to-many AlternativeFlows. Each flow of events has a PostCondition, which is a set of Sentences. There are three types of alternative flows: GlobalAlternative, SpecificAlternative, and BoundedAlternative. An alternative may have a condition, which is a sentence. A specific alternative flow corresponds to one step of the reference flow (either the basic flow or an alternative flow itself). A bounded alternative flow corresponds to more than one step of the reference flow. A global alternative flow is an alternative flow that refers to any step in the reference flow. As shown in Figure 11, the UCS of use case Withdraw Funds has a UseCaseSpecification that contains an instance of PreCondition, an instance of BasicFlow, an instance of BriefDescription, and three alternative flows (i.e., alt1, alt2, alt3). Actor ATM Customer is the primary actor of the use case.

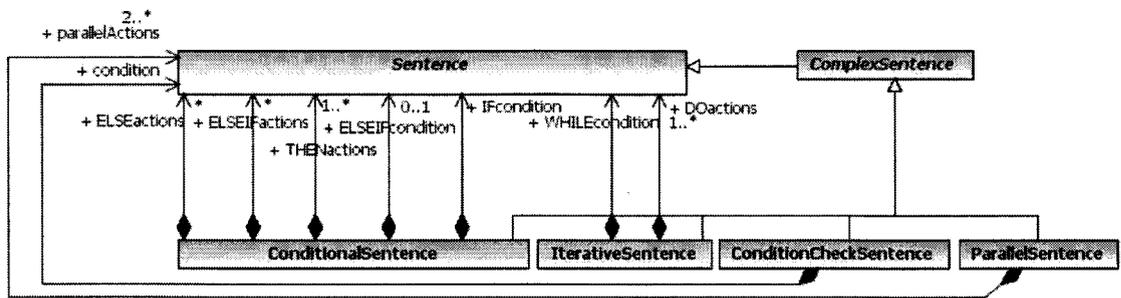


Figure 13 Package ComplexSentence of UCMeta

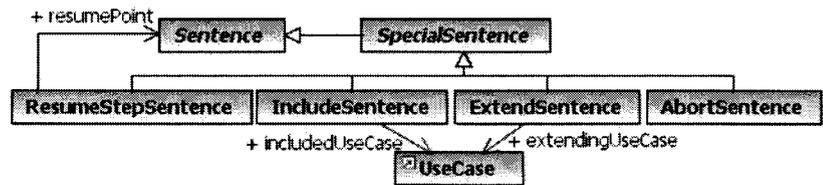


Figure 14 Package SpecialSentence of UCMeta

4.2.2 SentencePatterns

SentencePatterns is a package describing different types of simple sentence patterns. Sentence patterns help uniquely specify the grammatical structure of sentences. We classify sentence patterns into eight categories:

- 1) sv (subject-verb), e.g., Table 10, Specific Alt. Flow, step 2;
- 2) svc (subject-verb-complement), e.g., “The driver is added to the list of active drivers”;
- 3) svcc (subject-verb-complement-complement), e.g., “The system communicates with database to identify the number”;
- 4) svdo (subject-verb-direct object), e.g., Table 10, Basic flow, step 2;
- 5) svdoc (subject-verb-direct object-complement), e.g., “The system asks the customer for password”;

- 6) SVIODO (subject-verb-indirect object-direct object), e.g., “Customer sends the system a request”;
- 7) SVIODOC (subject-verb-indirect object-direct object-complement), e.g., “The system provides Sales an alternative part having enough stock”;
- 8) SLVSubjectComplt1 (subject-linking verb-subject complement), e.g., Table 10, Precondition.

This classification is a refinement of five basic English sentence patterns proposed in Linguistics (e.g., [13]) and English grammar books (e.g., [40]). We further refine each of these eight categories into sub-categories. For example, sentence pattern SVDO is further refined into *SysSubjVDO* (the subject is ‘the system’ of the UCMOD) and *ActorSubjDO* (the subject is an actor). We also refine sentence patterns containing object complements (i.e., SVC, SVCC, SVDOC, SVIODOC) and subject complements (i.e., SLVSubjectComplt) according to the different complement types they involve. For example, if the object complement of a SVDOC sentence is a prepositional phrase, then its sentence pattern is identified as *SVDOObjPPComplt* (subclass of class SVDOC). The complete taxonomy of sentence patterns is provided in Appendix C, along with a formal specification for each sentence pattern, including a brief description, a set of OCL expressions specifying constraints between elements of sentences, the sub-taxonomy, and examples. As shown in Figure 15, the condition of the condition check sentence *Basic flow step 8* (“the ATM has enough funds”) is a simple sentence with sentence pattern SVDO, with “the ATM” as subject, “has” as its predicator, and “enough funds” as its direct object.

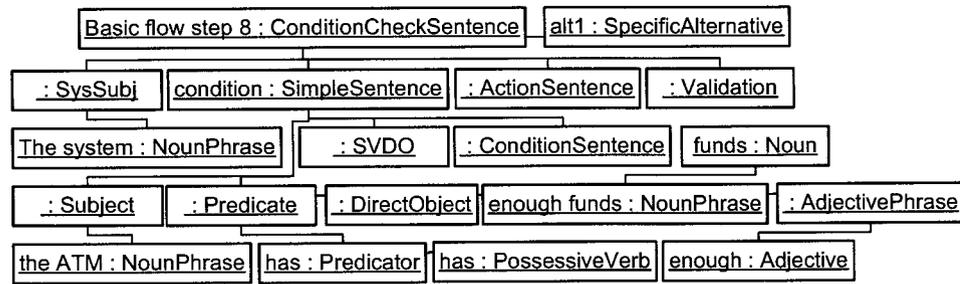


Figure 15 Example of package SentenceStructure

4.2.3 SentenceSemantics

SentenceSemantics is a package modeling the classification of sentences from the aspect of their semantic functions in a UCMOD. Each sentence in a UCS can either be a ConditionSentence or an ActionSentence (subclasses of abstract class Function). If the sentence is an action, then it must describe one of the following five transactions, of which the first four are reused from [21]:

- 1) Initiation: the primary actor sends a request and data to the system.
- 2) Validation: the system validates a request and data.
- 3) InternalTransaction: the system alters its internal state (e.g., recording or modifying something).
- 4) Response2PrimaryActor: the system replies to the primary actor with a result.
- 5) Response2SecondaryActor: the system sends requests to a secondary actor.

For example, Basic flow step 8 of use case *Withdraw Funds* is an Action sentence with Validation as its transaction type, as shown in Figure 15. The formalization of the five sentence semantics is presented in Appendix D for reference.

4.2.4 SentenceStructure

Package `SentenceStructure` takes care of NL concepts in sentences such as subject, object, verb, or Noun Phrase (NP). It is divided into three sub-packages: `Sentence`, `Phrase`, and `POS`. As shown in Figure 9, package `Sentence` depends on packages `Phrase` and `POS`. Package `Phrase` depends on package `POS`. This conforms to the hierarchical structure of sentences in NL: sentences are formed of phrases and words, and phrases are formed of words. As shown in Figure 16, package `Sentence` models the concepts of the top level constructs of a simple sentence (e.g., `Subject`). A predicate is composed of a `Predicator`, zero-to-many `Complement` (`SubjectComplt` or `ObjectComplt`), zero, one, or two `Objects`. There are two types of subject complements: NP complements (`SubjNPComplt`) and adjective phrase complements (`SubjAdjComplt`). Five types of `ObjectComplt` are considered in `UCMeta`: adjective phrase complements (`ObjAdjPComplt`), prepositional phrase complements (`ObjPPComplt`), present and past participle phrase complements (`ObjIngParticipleComplt` and `ObjEdParticiplePhrase`), and infinitive phrase complements (`ObjInfinitiveComplt`).

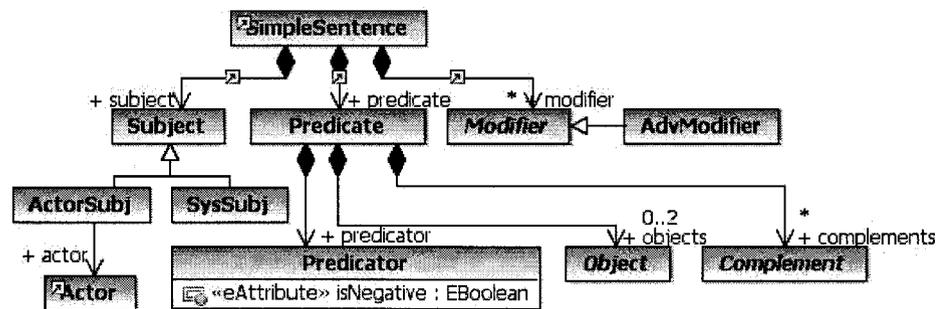


Figure 16 Package `SentenceStructure` of `UCMeta`

As shown in Figure 17, package `Phrase` specifies five types of phrases, conforming to the phrasal categories proposed in [13, 40]: `NounPhrase`, `VerbPhrase`, `AdjectivePhrase`, `AdverbPhrase`, and `PrepositionalPhrase`. `VerbPhrase` is further classified into `ParticiplePhrase`, `InfinitivePhrase`, and `GerundPhrase`. `ParticiplePhrase` has two types: `EdParticiplePhrase` and `IngParticiplePhrase`;

representing past participle phrase and present participle phrase, respectively. Each phrase is specified by following a specification pattern, which is composed of the head of the phrase, the pre-Head-Strings and post-Head-Strings of the phrase. These functional constituents [13, 40] are typed either by other phrases or parts of speech presented in package POS.

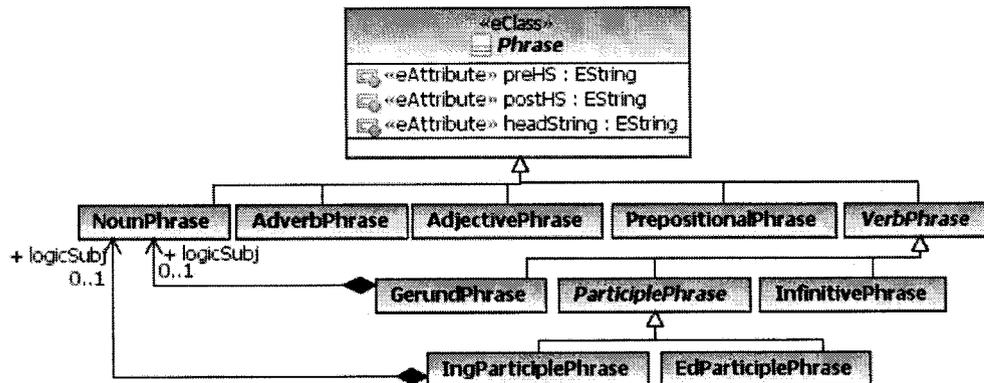


Figure 17 Package Phrase of UCMeta

For instance, as shown in Figure 18, the head of a noun phrase is a Noun. A noun phrase may take a Determiner, an AdjectivePhrase, a Noun, and/or another NounPhrase as pre-head-string. Its post-head-string can be a ParticiplePhrase, a PrepositionalPhrase, and/or an InfinitivePhrase. If a noun phrase contains an actor (or “the system”), then the position of the actor is specified as the attribute actorPosition (or sysPosition). For example, if a noun phrase is completely an actor, then the attribute takes the literal isActor of the enumeration NPActorPositionType as its value. Otherwise, if the noun phrase does not contain an actor, then the attribute takes the literal NounActor; if the contained actor is in the pre-head-string, then the attribute takes the literal preHeadStringActor; if the contained actor is in the post-head-string, then the attribute takes the literal postHeadStringActor; if the contained actor is the head of the noun phrase, then the attribute takes the literal headStringActor. The values of the attribute NPSystemPositionType are assigned by following the same logic. For example, an NP such as “a colorful paint on the wall” is composed of a determiner as a

pre-head-string (“a colorful”), an adjective phrase as another pre-head-string (“colorful”), a noun as head (“paint”), and a prepositional phrase as its post-head-string (“on the wall”).

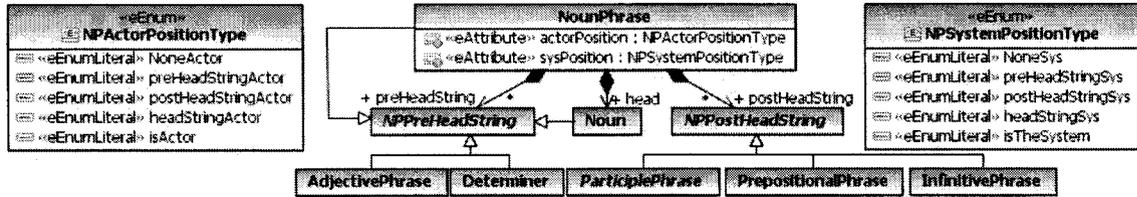


Figure 18 Metaclass NounPhrase and its functional constituents

Package POS describes seven parts of speech (also called word classes) of a sentence: Noun, Verb, Preposition, Adverb, Adjective, Conjunction, and Determiner. This classification is commonly used and recommended in Linguistics (e.g., [13]), English grammar (e.g., [40]), and NLP (e.g., [65]) books with little variation. For example, as shown in Figure 15, the condition (simple) sentence of step 8 in the basic flow of Table 10 has an NP (“the ATM”) as subject, its predicator is a possessive verb (“has”), and its direct object is an NP (“enough funds”) that has a noun (“funds”) as head and an adjective phrase (“enough”) as pre-head-string.

CHAPTER 5 **FORMALIZEUCM**

In this section, we discuss the transformation from the textual UCM_{od} to the intermediate model with respect to three aspects: its transformation rules (Section 5.1), algorithm (Section 5.2), and traceability (Section 5.3). We refer to this transformation as the FormalizeUCM activity in the remainder of the thesis.

5.1 Transformation rules

A UCM_{od} is composed of a use case diagram and a set of UCSs. The use case diagram is specified in UML: instances of the model elements of package `UML::UseCases`; therefore for this part no transformation is required. The rest of the section follows the structure of the UCM_{eta} architecture (Figure 10) to present the transformation rules.

5.1.1 UCM_{eta}, UML::UseCases and UCSTemplate

By default, an instance of metaclass `UseCaseModel` of package UCM_{eta} is created to contain all instances of metaclass `UseCaseModelElement` and its subclasses, including `UML::UseCases`, as previously discussed in Section 4.2.

There is no required transformation for package `UML::UseCases`, since it is specified in UML. However, it should be noted that a link between a use case and its corresponding UCS should be established (instance of the association between metaclasses `UML::UseCases::UseCase` and `UseCaseSpecification`), such as the link between `Withdraw Funds::UseCase` and `:UseCaseSpecification` in Figure 11.

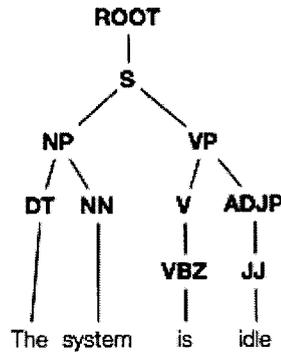
Each field of a textual UCS (Table 10) is transformed into its corresponding model element of package UCSTemplate. The transformation is a straightforward one-to-one mapping. The keywords specifying restrictions in textual UCSs map to different types of `SpecialSentences` and `ComplexSentences`. For example, basic flow step 8 of Table 10 (containing keyword `VALIDATES THAT`) is transformed into an instance of `ConditionCheckSentence`, a type of complex sentence, as shown in Figure 11.

5.1.2 SentenceStructure

Sentence structures are identified by a NL parser, the Stanford Parser [101]. Given a sentence, the parser outputs its parse tree (with specific node types indicating syntactic constituents of the sentence) and the grammatical dependencies of the sentence that describe the relations of functional roles in the sentence (e.g., the dependency between the subject and predicate). Our transformation further transforms these two outputs into instances of the metaclasses in package `SentenceStructure`.

The transformation attempts to match a sub-path of the parse tree with a specific sequence of types of nodes with dependency information to create those UCMeta instances. For example, Figure 19 presents the parse tree and the three dependency relations provided by the Stanford Parser for sentence ‘The system is idle’. The predicate of the sentence (‘is idle’) is identified when the parse tree of the sentence contains ROOT-S-VP, where ROOT denotes the root of the parse tree, S (the syntactic tag denoting ‘Sentence’) is the sub-tree of the ROOT tree, and VP (the syntactic tag denoting ‘Verb Phrase’) is one of the sub-paths of the parse tree. Notice that the identification of the predicate of a sentence does not need the dependency information. However, identifying subjects cannot be easily achieved just by analyzing the parse tree. Analyzing the dependency information is also required. For example, the second dependency shown in Figure 19 is with *nsubj* type, which denotes the dependency between the subject and predicator of the sentence. With this dependency information, we can identify that the main part of the subject of the sentence is ‘system’. Now, we need to identify the sub-tree of the parse tree that represents the subject of the sentence. In this particular example, the sub-tree we are looking for is ROOT-S-NP, where NP is the noun phrase that forms the subject (i.e., ‘The system’). This is done through iterating over each leaf contained in the sub-tree (e.g., ‘NN’ denoting Noun) to find a leaf with the value that matches the value of the element ‘system’ of the *nsubj* dependency and then identify the sub-tree by tracing back from the leaf node (i.e., ‘NN’) to its parent (e.g., ‘NP’). By further analyzing the sub-tree and the first dependency shown in Figure 19, we can then know that the subject can be decomposed into the determiner (‘The’) and the noun (‘system’).

Parse tree:



Dependencies:

1. det(system, The)
2. nsubj(idle, system)
3. cop(idle, is)

Figure 19 An example of parse tree and dependencies

One can see that it is not trivial to relate the syntactic structure of a sentence represented as a parse tree (e.g., noun phrase) to its functional roles specified in the dependency information (e.g., subject) so that corresponding model elements in package SentenceStructure can be correctly populated. To do so, a complex algorithm (Section 5.2) is required. In certain NLP contexts, this relation might not be important. However, we need to generate an integrated intermediate model that captures all the information of a UCMOD, to facilitate the subsequent transformation (GenerateUML) (Chapter 6).

5.1.3 SentencePatterns and SentenceSemantics

The identification of sentence patterns and sentence semantics is performed once the other transformations have been achieved, i.e., it relies on an existing UCMeta instance, identifying patterns of metaclass instances and relationships. For example, a sentence conforms to the `SVDO` pattern if it has one subject, one action verb (the other possible kind of verb is called a linking verb) as predicator, one direct object, without any complement. As an example, the formal specification/identification in OCL of the sentence pattern `SVDO` is presented in Figure 20. As another example, if the subject of a simple sentence is an actor, that actor is the primary actor of the use case containing this sentence, and either the direct object or complements of the sentence contains 'the system', then the sentence

has a high probability of describing an `Initiation` (sentence semantics) where the primary actor sends a request and data to the system: this is the case of sentence “the customer requests the system to display account information.” The formal specification/identification of `Initiation` is given in Figure 21.

```

Context SimpleSentence
self.sentencePattern.OclIsTypeOf(SVDO) implies
  self.predicate.precicator.form.head.OclIsTypeOf(ActionVerb)
  and self.predicate.object->size()=1
  and self.predicae.object->exists(OclAsType(DirectObject))
  and self.predicate.complement->size()=0

```

Figure 20 Formal specification/identification of svdo

```

Context Sentence
sentence.functionType.OclIsTypeOf(ActionSentence) and
self.transactionType.OclIsTypeOf(Initiation)
implies
  self.OclAsTypeOf(SimpleSentence).subject.OclIsTypeOf(ActorSubj)
  and
  (
    (sentence.predicate.objects->size()>0 and
      sentence.predicate.objects->exists
        (obj|obj.OclIsTypeOf(DirectSysObj))
    )
  or
    (sentence.predicate.complements->size()>0 and
      sentence.predicate.complements->exists
        (complt|complt.OclIsTypeOf(ObjPPComplt) and
          complt.content.isContainingSystem()
        )
    )
  )
)

```

`isContainingSystem()` is used to check whether the content of the complement of the sentence contains the string “the system”.

Figure 21 Formal specification/identification of Initiation

The `SentencePatterns` package (Section 4.2.2) of `UCMeta` is particularly designed to group queries on the syntactic constituents and functional roles of a sentence (e.g., query on the subject of a sentence), frequently involved in the transformation rules of the transformation `GenerateUML` (Chapter 6). By doing so, the efficiency of the transformation of `GenerateUML` can be largely improved. The `SentenceSemantics` package (Section 4.2.3) captures the semantic meaning (e.g., condition sentence, initiation sentence) of a sentence in the context of use cases. This kind of semantic

information particularly helps simplify the generation of sequence diagrams (Chapter 8); therefore the transformation rules for generating sequence diagrams (Table 13) can be well-structured and the efficiency of the transformation can be improved.

5.2 Algorithm

As UCMeta is layered and partitioned, it is natural for our transformation algorithm to follow a nested structure. Operation *ParseUCM* (Figure 22) is the top operation which invokes operation *ParseUCS* to parse each UCS. Operation *ParseUCS* further invokes operation *ParseSimpleSentence* to parse each simple sentence, which further invokes other operations to parse predicates, subjects, etc.

Parsing a UCS consists in parsing its structure (i.e., template), thereby identifying actors, brief description, flows, and the other template fields. The rules presented in Section 5.1.1 are used. While parsing flows, complex sentences (e.g., with a keyword like VALIDATES THAT) and special sentences (e.g., with a keyword like ABORT) are identified. Eventually, only simple sentences remain to be parsed and we rely on an off-the-shelf NL parser for that task [101]: rules of Section 5.1.2 are used. Then, sentence patterns and semantic types are identified (rules in Section 5.1.3).

5.2.1 ParseUCM and ParseUCS

These two operations are presented together as an activity diagram in Figure 22. The transformation starts when the UCMMod being parsed is imported. Parsing a UCS contains two sequential steps: *Parse use case template* and *Parse each simple sentence*. The first step contains three sequential sub-steps: *Parse the UC template* (applying transformation rules described in Section 5.1.1), *Parse complex sentences* (identifying keywords like VALIDATES THAT) and *Parse special sentences* (identifying keywords like ABORT). The second step contains five sequential sub-steps. The first sub-step is to invoke the functions provided in the NL parser to obtain the parse tree of the sentence. Then, the dependencies of the parts of speech (describing the grammatical relations of POS) in the sentence are obtained from the NL parser. With the parse tree and the dependencies, the

simple sentence is parsed and transformed into instances of the model elements of package `SentenceStructure`. After each simple sentence is parsed, the algorithm identifies its sentence pattern. The last step is to identify the semantic type of each sentence.

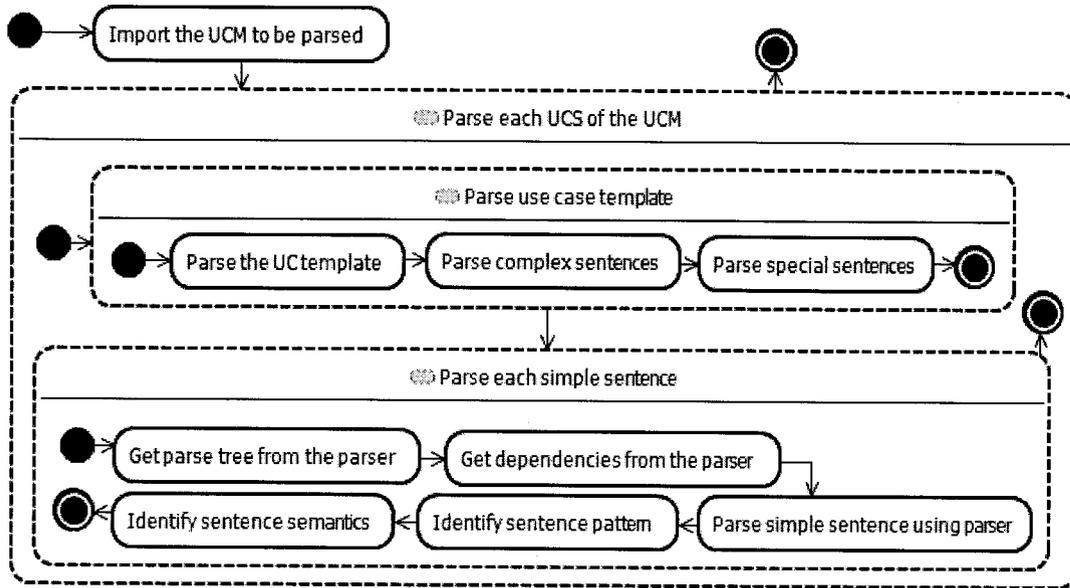


Figure 22 ParseUCM operation

5.2.2 ParseSimpleSentence

As shown in Figure 23, an activity diagram is used to present the algorithm of parsing a simple sentence. The first two steps of parsing a simple sentence involve obtaining the parse tree and the dependencies of the sentence from the NL parser. The algorithm iterates over each child tree of the parse tree. If there is a child tree that has the syntactic tag ‘ADVP’, then the child tree is identified as the adverb modifier of the sentence. If there is a child tree with the syntactic tag ‘VP’, then the child tree is identified as the predicate of the sentence. The operation *ParseSimpleSentence* is composed of a series of operations to identify the subject, predicate, predicator, and modifiers of a sentence. The algorithm in detail (pseudocode) is provided in Appendix E.

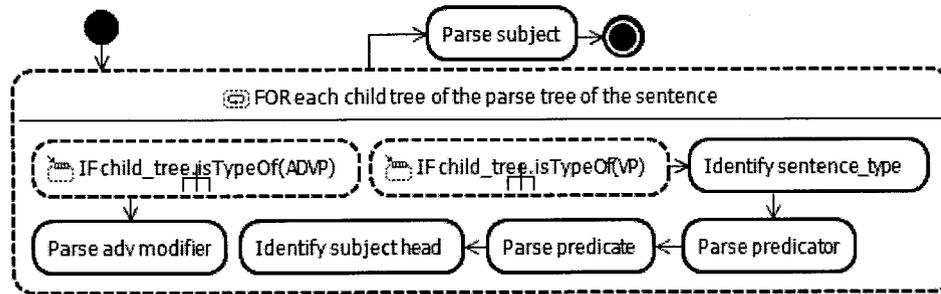


Figure 23 ParseSimpleSentence operation

5.3 Traceability

When deciding which traceability links to identify and record, one needs to find a tradeoff between the effort to record them and maintain them, and the benefits in terms of supporting change impact analysis, change assessment and management, and artifacts comprehension. In our context, one set of traceability links from textual UCSs to formalized UCSs (UCMeta) is straightforward: Fields of the use case template used to document textual UCSs are simply linked to the corresponding metaclasses in UCMeta during transformations. For example, field *Brief Description* of the use case template is linked to metaclass `BriefDescription` of UCMeta. A sentence in the brief description is then linked to metaclass `Sentence` of UCMeta. These links are simple to establish and maintain. The question is then whether it is worth having more detailed links. For instance, a step in a flow could be linked to its constituents (e.g., subject) in the UCMeta instance. With finer-grain links like those more detailed impact analysis could be performed for instance. However, to maintain such links when the UCS changes would require re-parsing the whole sentences, which defeats the purpose since then one can establish new links right away: no time or effort would be saved. Research conducted by Egyed et al. also indicates, based on the analysis results of three case studies, that “it can be worthwhile to reduce the level of detail during tracing to save effort while the loss in quality might still be acceptable” [30]. So we believe that it is not worth creating more detailed traceability links than the straightforward ones mentioned earlier, and those

simple links seem sufficient to perform typical traceability-based analyses, such as impact analysis. Future work will investigate whether more detailed links are necessary.

CHAPTER 6 GENERATEUML

GenerateUML transforms a formalized UCMOD (i.e., an instance of UCMeta) into a UML analysis model (i.e., an instance of the UML 2 metamodel). We classify GenerateUML transformation rules into four categories: transformation rules for generating the overall structure of the analysis model (rule set A), transformation rules for generating a class diagram (rule set B), transformation rules for generating sequence diagrams (rule set C), and transformation rules for generating activity diagrams (rule set D). In this section, we discuss rule set A (Section 6.1), the overall algorithm of generating analysis models (Section 6.2), and the traceability model applied (Section 6.3). The detailed discussion of the generation of class, sequence and activity diagrams are presented in Chapter 7, Chapter 8, and Chapter 9, respectively.

6.1 Rule Set A: Structure

This rule set is used to generate the overall structure of an analysis model. It relies on a class taxonomy, originally specified by Jacobson et al. [51] and now part of modeling practice [44] to specify class responsibilities: Boundary objects handle interaction between the actors and the system; Entity objects are responsible for storing and providing access to data; Control objects controls the interaction of participating objects during the execution of a use case. In UML, these types of classes are specified with stereotypes <<Boundary>>, <<Entity>>, and <<Control>>.

In rule set A, an abstract <<Control>> class is generated for the UML model (rule A0), with the same name as the model. It is the superclass of the <<Control>> classes generated for each use case (rule A1, also conforming to one of the heuristics proposed in [14]). A <<Boundary>> class is generated for each actor (rule A2). If a use case specializes another use case, then the control class corresponding to the more specific use case specializes the control class corresponding to the more general use case (rule A3). The relationship between an actor and a use case is transformed into an association between the control class corresponding to the use case and the boundary class

corresponding to the actor (rule A4). A generalization between two actors is transformed into a generalization between the corresponding boundary classes (rule A5). Rule A6 is used to generate an instance of `Interaction` for each use case.

The rules have some dependencies, thereby specifying orders of rules execution: Rules A1, A2 and A6 depend on the execution of rule A0; A3 depends on the execution of A1; A4 must execute after A1 and A2; A5 executes after A2.

6.2 Algorithm

Transformation rule sets A, B, C and D are applied sequentially. Within each set, some sequential constraints exist between rules (recall Section 6.1). Rules at the same level are independent and can be executed in any sequence. However, since all the rules in categories B, C and D are hierarchical—one rule may compose other rule(s), composition relationships should be satisfied when the rules are executed. Structuring rules this way facilitates the modification, addition, and deletion of rules and also simplifies the algorithm.

6.3 Traceability

Traceability links are established between a formalized UCMOD and the analysis model generated from it while `GenerateUML` is performed. We adapted the traceability model proposed in the traceability component (`fr.irisa.triskell.traceability.model`) of `Kermeta` [55], by adding two subclasses (`UCMReference` and `UMLReference`) to the abstract class `Reference` and simplifying class `Message` by only keeping one attribute `value`, as shown in Figure 24. A `TraceModel` is composed of a set of `Traces`, `Messages`, and `References`. A trace connects a UCMOD model element (source), i.e., an instance of `UCMReference`, to a UML model element (target), i.e., an instance of `UMLReference`. When a transformation rule is executed, one or more traces are created to link the source model element(s) and the target model element(s). The rationale for establishing the links, i.e., the transformation rule being used, is recorded in attribute `value` of class `Message`.

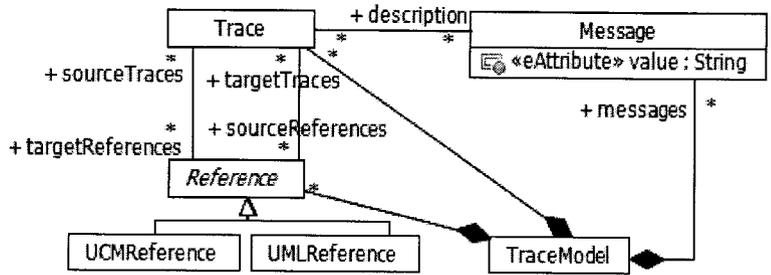


Figure 24 Traceability model

CHAPTER 7 GENERATE CLASS DIAGRAMS

In this section, we discuss how aToucan automatically derive an analysis class diagram from a use case model. We present the transformation rules (rule Set B) in Section 7.1, followed by the algorithm (Section 7.2).

7.1 Transformation rules

Rule set B is further decomposed into two subsets: B1 and B2, which are executed sequentially. Most of these rules implement heuristics proposed by Abbott [3] and Software Engineering textbooks (e.g., [14, 60]), while others are based on our own experience of deriving class diagrams from requirements. The automatically generated class diagram for the ATM system is partially provided in Figure 25 and Figure 26. All the entity classes are presented in Figure 26, where association names are omitted for layout purpose. The control class `Withdraw Funds` and its associated classes are presented in Figure 25, where the attributes and operations of the entity classes are omitted.

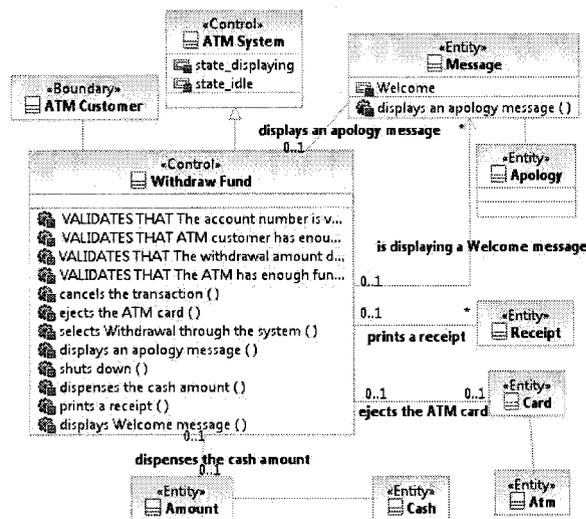


Figure 25 Class diagram automatically generated by aToucan for the ATM system (Withdraw Fund Control class)

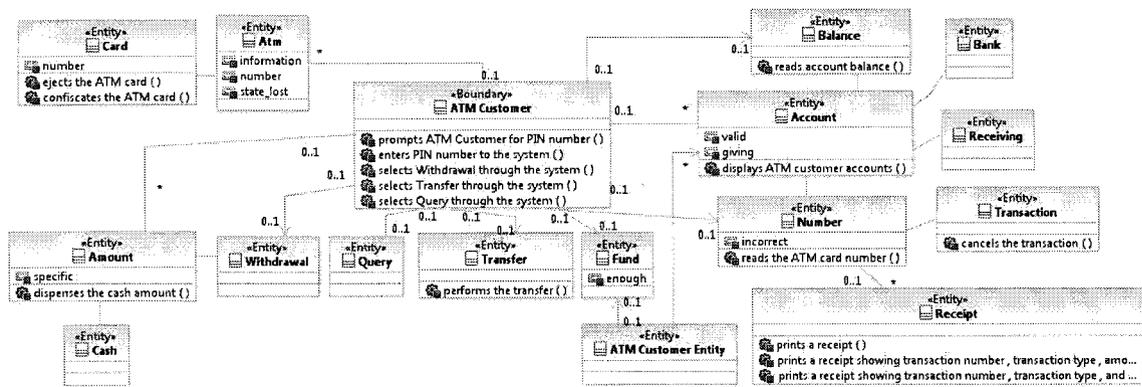


Figure 26 Class diagram automatically generated by aToucan for the ATM system (entity classes)

Seventeen rules, summarized in Table 11 (B1), aim to identify entity classes, a subset of the attributes and operations in these classes, and a subset of the class associations by analyzing NPs. It consists of four main rules: B1.1-B1.4. Rules B1.1 and B1.2 identify entity classes from non-possessive and possessive NPs, respectively, and should execute before rules B1.3 and B1.4. Rules B1.3 and B1.4 are further composed of five third level rules, which identify attributes, operations, and associations by analyzing pre- and post-heading strings of NPs when these NPs contain an actor (rule B1.4) or not (rule B1.3). Rules B1.3 and B1.4 rely on seven other rules to handle different types of pre- and post-heading strings. Note that the numbering of the transformation rules is based on their hierarchical levels. For example, rule B1.1 is a second level rule and therefore it can only be contained by a first level rule. The first number of B1.1 indicates which is higher level rule containing it (i.e., B1).

Let us use a couple of examples to illustrate some of the transformations performed by these rules. As discussed in Section 4.2.4, a NP is composed of three constituents: the mandatory head, an optional pre-head-string, and an optional post-head-string. If a NP (e.g., “the account” in Step 6, Basic flow, Table 10) has a noun (“account”) as its head with only a determiner (“the”) as its pre-head-string, then the NP is transformed into an entity class (e.g., Account, Figure 26) by rule B1.1.

Otherwise, we have to further analyze the pre- and post- head strings to identify attributes, operations, and associations. For example, one rule contained in rule B1.3 specifies that when one of the pre-head-strings of a NP is a noun and that noun has been transformed into an entity class, say class A, during a previous transformation, and the head of the NP or the NP itself has also been transformed into an entity class—all this constitutes the pre-condition for rule B1.0.0.1 (composed by rule B1.0.1, which is composed by B1.3), then an association is established between these two entity classes. For example, consider NP “the account number”, which pre-head-string is “account”. Assuming classes `Account` and `number` already exist, applying the rule adds an association between them, as shown in Figure 26.

Thanks to these rules in set B1, we make sure that each NP is systematically analyzed and its natural language information is well utilized to generate the model elements in the class diagram.

Based on the results of rule sets A and B1 (i.e., a partially generated analysis model), rule set B2 (Table 12) identifies additional attributes, operations and associations by further analyzing sentences. B2 is composed of eleven rules (B2.1-B2.11), which handle different types of sentence patterns and function types (condition or action sentences) (Section 4.2.2 and Section 4.2.3). For example (precondition in Table 10), the sentence “The system is idle” has a verb that links a subject and a subject complement, i.e., it is a subject-linking verb-subject complement (which is represented by metaclass `SLVSubjectComplt`): its subject is “the system”, “is” is the linking verb, and “idle” is the subject complement. When such a sentence is identified, rule B2.1 transforms the subject complement (“idle”) into an attribute (attribute `idle`) of the class representing the subject, thereby indicating one of the states of that class. In this example (from Table 10), the class representing the “the system” subject, called `Withdraw Funds after the use case` name, has been created by rule A1 (as shown in Figure 25): recall that this rule creates a `<<Control>>` class for each use case to represent “the system”, and executes before rules in set B. For example, as shown in Figure 25, attribute `state_idle` was originally

generated for each control class corresponding to each use case. Then it was pulled up to the super control class `ATM System`. Rule B2.6 further relies on five rules to handle different types of object complements: for instance, a prepositional phrase can be the object complement of a sentence (metaclass `ObjPPComplet`). This set of rules can be considered complete in the sense that it complies with the taxonomy of sentence patterns and the taxonomy of subjects and object complements specified in `UCMeta` (Section 4.2.2), which itself is complete since it is derived from the literature on linguistics.

We also specified rules to determine association multiplicities, which are not described in Table 11 and Table 12. The multiplicity of the end of an association depends on the type the article (indefinite ('a' or 'an') or definite ('the')) of the noun or NP where the entity class connected to the end of the association is derived. More specifically, if the article is an indefinite article, then multiplicity zero-to-many ('*') is assigned to the association end; otherwise, multiplicity zero-to-one ('0..1') is assigned. The multiplicity of an association end can also be determined by the form (singular or plural) of the noun or NP; plural nouns or NPs imply multiplicity zero-to-many ('*') while singular ones indicates multiplicity zero-to-one ('0..1'). These two general rules are used by rules in rule set B when associations are generated. However, there exist several cases, where different rules should be applied. When rules B1.0.0.1, B1.0.0.3 and B1.0.0.5 (Table 11) are applied to generate an association, both ends of the association are assigned multiplicity one ('1'). For example, as shown in Figure 26, both ends of the association between classes `Account` and `Balance` have multiplicity one ('1'), which are derived from NP "account balance" in the UCSs using rule B1.0.0.1. In terms of determining the navigation of an association, all the rules except B1.0.0.1 and B2.6.3 generate unidirectional associations. The navigable end of a unidirectional association being generated by each of these rules is indicated in Table 11 and Table 12.

Table 11 Summary of rule Set B1

Rule #	Description	Type	Composed rules
B1	Process noun phrases to identify entity classes, associations, attributes and operations.	Composite	B1.1-B1.4
B1.1	When a NP is a single noun or a noun with a determiner, an entity class is generated to represent the NP.	Atomic	
B1.2	When a NP contains a possessive noun, an entity class is generated for the possessive noun.	Atomic	
B1.3	Process noun phrases not containing actors.	Composite	B1.0.1-B1.0.4
B1.4	Process noun phrases containing actors.	Composite	B1.0.1, B1.0.2, B1.0.5
B1.0.1	Process the pre-head-string of a NP that itself or its head has been identified as an entity class.	Composite	B1.0.0.1, B1.0.0.2
B1.0.2	Process the post-head-string of a NP that itself or its head has been identified as an entity class.	Composite	B1.0.0.3, B1.0.0.4
B1.0.3	Process the pre-head-string of a NP that itself or its head has not been identified as an entity class.	Composite	B1.0.0.2, B1.0.0.6, B1.0.0.7
B1.0.4	Process the post-head-string of a NP that itself or its head has not been identified as an entity class.	Composite	B1.0.0.3, B1.0.0.4
B1.0.5	When the pre-head-string of a NP is an actor (boundary class), an association is generated between the boundary class and the entity class representing the head of the NP if there exists such an entity class; otherwise, the head of the NP is transformed into an attribute of the boundary class.	Atomic	
B1.0.0.1	When the pre-head-string of a NP is a noun and has been transformed into an entity class and the NP has been transformed into an entity class, an association is established between these two entity classes.	Atomic	
B1.0.0.2	When the pre-head-string of a NP (entity class) is an adjective phrase, it is transformed into an attribute of the entity class.	Atomic	
B1.0.0.3	When the post-head-string of a NP (entity class) is a verb phrase, an operation is generated for the entity class and an association is generated between the entity class and another entity class representing the object of the verb phrase if such an entity class exists (B1.0.0.5).	Composite	B1.0.0.5
B1.0.0.4	When the post-head-string of a NP (entity class) is a prepositional phrase, an association is generated between the entity class and another entity class representing the object of the prepositional phrase if such an entity class exists; otherwise generate a new entity class to represent the object of the prepositional phrase (B1.0.0.7).	Composite	B1.0.0.7
B1.0.0.5	When the pre-head-string of a NP (entity class) is another NP, an association is generated between these two entities.	Atomic	
B1.0.0.6	When the pre-head-string of a NP is a noun and it has been identified as an entity class, the head of the NP is identified as an attribute of the entity class.	Atomic	
B1.0.0.7	When the pre-head-string of a NP is a noun and it has not been identified as an entity class, an entity class is created for the noun and an association is established between the entity class and another one representing the NP if such an entity class exists.	Atomic	

Table 12 Summary of rule Set B2

Rule #	Description	Type ¹³	Composed rules
B2	Process each sentence to identify attributes, operations, and associations.	C	B2.1-B2.11
B2.1	SLVsubjectCompLit: subject complement → state attribute of the subject class.	A	
B2.2	Condition sentence, SVDO or SVC: predicate → state attribute of the subject class.	A	
B2.3	SV, non-passive sentence: predicate → operation of the subject class.	A	
B2.4	SVDO: if the predicator is a possessive verb, transform the object into an attribute of the subject class; otherwise, if the object class exists, transform the predicate into an operation of the object class, and if the object class does not exist, transform the predicate into an operation of the subject class. Generate an association between the subject class and the object class if both classes exist.	A	
B2.5	SVIODO: invoke B2.4 and generate an association between the subject class and the indirect object class if both classes exist.	C	B2.4
B2.6	SVIODOC and SVDOC: invoke B2.4 or B2.5, and invoke B2.6.1-B2.6.5 to process different types of object complements.	C	B2.4, B2.5, B2.6.1-B2.6.5
B2.6.1	SVDOobjPPCompLit and the preposition of the PP is “for”: transform the predicate into an operation for the direct object class and transform the object of the PP into a parameter of the newly generated operation.	A	
B2.6.2	SVDOobjPPCompLit and the prepositions are others except “for” and “of”: transform the predicate into an operation of the direct object class, generate a parameter named as the direct object for the operation, and establish an association between the subject class and the PP object class.	A	
B2.6.3	SVDOobjIngParticipleCompLit: transform the complement into an operation of the direct object class and establish an association between the direct object class and the complement object class.	A	
B2.6.4	SVDOobjEdParticipleCompLit: transform the complement into an operation of the direct object class and establish an association between the direct object class and the complement object class.	A	
B2.6.5	SVDOobjInfinitiveCompLit: transform the complement into an operation of the direct object class and establish an association between the direct object class and the complement object class.	A	
B2.7	SV and passive sentence: transform the predicate into an attribute of the subject class.	A	
B2.8	SVC and passive sentence: transform the predicator into an operation of the subject class; if the complement is a PP and its preposition is “by”, then establish an association between the subject class and the PP object class.	A	
B2.9	SVCC: if one of the complements is a PP and the other is an infinitive phrase, then transform the infinitive phrase into an operation of the PP object class and transform the object of the infinitive phrase into a parameter of the operation; if one of the complements is a PP, then establish an association between the subject class and the PP object class.	A	
B2.10	ConditionCheckSentence: generate an operation for the subject class with the name “VALIDATES THAT” and transform the condition of the sentence into a parameter of the operation; if there is an actor involved in the sentence, an association is established between the subject class and the actor class.	A	
B2.11	SVC and no passive sentence: if the complement is a PP, then transform the complement into an attribute of the subject class.	A	

¹³ ‘C’ denotes ‘Composite’, while ‘A’ denotes ‘Atomic’.

7.2 Algorithm

The algorithm for generating class diagrams are provided in Figure 27 to Figure 29. Figure 27 presents the high-level algorithm for transforming a UCMod (an instance of UCMeta) into a class diagram contained in the corresponding analysis model (an instance of the UML 2.0 metamodel). The `Transform_CD` function has one input parameter (i.e., `ucModel:ucm::UseCaseModel` - the use case model being transformed into the class diagram) and one output parameter (i.e., `umlModel:uml::Model` - the UML model transformed from the UCMod). The algorithm first collects all noun phrases contained in each use case specification of the UCMod. Then it sequentially invokes rules in rule sets B1 and B2 (Section 7.1) to generate class diagram model elements. By generating class diagrams based on the information of all the use cases, we make sure that we utilize all the information of UCMod in a systematic way and therefore generate the most complete class diagrams possible.

Operations `invoke_rule_B1()` and `invoke_rule_B2()` correspond to the composite rules B1 and B2 in Figure 28 and Figure 29, respectively. Operation `invoke_rule_B1()` further invokes rules B1.1-B1.4 to identify entity classes, associations, attributes and operations from all NPs, as shown in Figure 28. These four composite rules further invoke other rules contained in them, which are omitted from the algorithm in Figure 28. As shown in Figure 29, operation `invoke_rule_B2()` invokes different types of rules according to the types of sentences. For example, if a simple sentence is a `SVDO` action sentence, then rule B2.4 is invoked to generate corresponding model elements (Figure 29).

Algorithm **Transform (ucModel:ucm::UseCaseModel) : umlModel**
Input ucModel : ucm::UseCaseModel --- The UCMOD being transformed into a class diagram
Output umlModel : uml::Model --- The UML model transformed from the UCMOD
Declare allNPs : Set(ucm::NounPhrase) --- all noun phrases contained in the UCMOD
Begin
 1. allNPs ← getAllNounPhrases (ucModel)
 2. invoke_rule_B1(allNPs)
 3. invoke_rule_B2(ucModel)
 4. **return** umlModel
End

Figure 27 Transformation algorithm for generating class diagrams

Operation invoke_rule_B1(allNPs:Set(ucm::NounPhrase))
allNPs.each {np : ucm:: NounPhrase
 if (np.equals(ucm::UseCaseNL::NPSystemPostitionType.NoneSystem)
 and np.equals(ucm::UseCaseNL::NPActorPositionType.NoneActor)) **then**
 if (np.preHeadString.size ==0 **or** (np.preHeadString.size ==1 **and**
 np.preHeadString.one.isInstanceOf(ucm::Determiner)) **and** np.postHeadString.size()==0)
 then
 invoke_rule_B1.1(np)
 else
 if (np.preHeadString.size>0 **and** np.preHeadString.exists {preHead |
 preHead.isInstanceOf(PossessiveNone)})
 then
 invoke_rule_B1.2(np)
 else
 invoke_rule_B1.3(np)
 end
 end
 end
 if not np.actorPosition.equals(UseCasesNL::NPActorPositionType.NoneActor) **then**
 invoke_rule_B1.4(np)
 end
} }
End

Figure 28 Transformation algorithm for generating class diagrams -

invoke_rule_B1

Operation invoke_rule_B2(ucModel:ucm::UseCaseModel)
sens : Set(ucm::SimpleSentence)
csens : Set(ucm::ComplexSentence)
ucModel.modelElements.select {ele | ele.isInstanceOf(ucm::UseCase)}.each {uc |
 sens ← getAllSimpleSentences(uc.specification)
 csens ← getAllComplexSentences(uc.specification)
 sens.select {sen|sen.functionType.isInstanceOf(ucm::ConditionSentence)}.each {csen|
 if csen.container.isInstanceOf(PreCondition) **or** csen.container.isInstanceOf(PostCondition)
 then
 if csen.pattern.isInstanceOf(SVDO) **then** invoke_rule_B2.2(csen, uc.specification) **end**
 end
 if csen.pattern.isInstanceOf(SVC) **and** csen.pattern.isPassiveSentence **then**
 invoke_rule_B2.2(csen, uc.specification)
 end

```

}
sens.each{sen|
  if sen.pattern.isInstanceOf(SLVSubjectComplt) then
invoke_rule_B2.1(sen, uc.description) end
  if sen.pattern.isInstanceOf(SV) then
    if sen.pattern.isPassiveSentence then
      invoke_rule_B2.7(sen, uc.description)
    else
      invoke_rule_B2.3(sen, uc.description)
    end
  end
  if sen.pattern.isInstanceOf(SVDO) then invoke_rule_B2.4(sen, uc.description) end
  if sen.pattern.isInstanceOf(SVIODO) then invoke_rule_B2.5(sen, uc.description) end
  if sen.pattern.isInstanceOf(SVDOC) or sen.pattern.isInstanceOf(SVIODO) then
    invoke_rule_B2.6(sen, uc.description) end
  if sen.pattern.isInstanceOf(SVC) then
    if sen.pattern.isPassiveSentence then invoke_rule_B2.8(sen, uc.description)
    else invoke_rule_B2.11(sen, uc.description) end
  end
  if sen.pattern.isInstanceOf(SVCC) then invoke_rule_B2.9(sen, uc.description) end
}
csens.select{sen|sen.isInstanceOf(ucm::ConditionalCheckSentence)}.each{sen |
invoke_rule_B2.10(sen, uc.description)}
}

```

End

**Figure 29 Transformation algorithm for generating class diagrams -
invoke_rule_B2**

CHAPTER 8 GENERATE SEQUENCE DIAGRAMS

In this section, we discuss how a Toucan automatically derive analysis sequence diagrams from a UCM_{od}. The transformation rules are discussed in Section 8.1 and the algorithm is presented in Section 8.2.

8.1 Transformation rules

The transformation from an instance of UCM_{eta} to sequence diagrams involves 18 rules, summarized in Table 13. Note that these transformations rely on the identification of classes and operations of a class diagram, as previously discussed in Chapter 7. The indentation of rules in Table 13 indicates some rules (composite rules) are composed of, and therefore invoke other rules (atomic rules), e.g., composite rule C1 invokes atomic rules C1.1-C1.6. Rules C1-C3 process three types of sentences: `SimpleSentence`, `ComplexSentence`, and `SpecialSentence`.

Rules C1.1-C1.6 process seven of the eight different types of simple sentences, e.g., `SV`, `SVC` (Section 4.2.2). `SLVSubjComplt` (subject-link verb-subject complement) sentences (e.g., “The system is idle”) are condition sentences and are therefore not included in this rule set. These sentences appear in pre- and post-conditions, or in complex sentences (e.g., condition check sentences and conditional sentences handled by rules C2.1 and C2.2, respectively).

Rule C1.1 generates a self-message on the system lifeline for sentences with pattern `SV`. These sentences all have type `InternalTransaction` and their subject is therefore always “the system”. We generate a lifeline in each sequence diagram to represent “the system”, which we refer to as “the system lifeline” in the rest of this paper. Sentences with patterns `SVC`, `SVCC`, `SVDO`, `SVDOC`, `SVIODO`, and `SVIODOC`, handled by rules C1.2-C1.6, might have four different transaction types: `Initiation`, `InternalTransaction`, `ResponseToPrimaryActor`, and `ResponseToSecondaryActor` (Section 4.2.3). Each rule

generates different UML sequence diagrams elements according to this transaction type. For example, if the sentence pattern of a simple sentence is `SVDO` and its transaction type is `Initiation`, then rule C1.4 generates two sequential messages: one from the actor lifeline to the actor boundary lifeline and one from the boundary lifeline to the system lifeline. The rationale is that the primary actor sends a request and data to the system in an `Initiation` sentence (Section 4.2.3) and therefore the subject of the sentence is always the primary actor of the use case and its direct object must be the system. The condensed notation “actor lifeline (subject) → actor boundary lifeline → the system lifeline (direct object)” in Table 13 represents these two sequential messages. If the transaction type of the sentence is `InternalTransaction` then only one message from the system lifeline to the lifeline representing the class identified from the object of the sentence (Section 4.2.3) is generated. Notice that if such a lifeline does not exist, it is created (the classifier is the class derived from the object of the sentence) and a create message from the system lifeline to this newly created lifeline is generated. For example, the `InternalTransaction` sentence (i.e., basic flow, step 9, in Table 10) includes object “the cash amount”, which is a noun phrase with noun “amount” as its head. This noun is identified as a class (Figure 26). According to rule C1.4, a lifeline representing entity class `Amount` (i.e., `<<Entity>>:Amount`) is generated and a message (i.e., `8: dispenses the cash amount`) from the system lifeline (i.e., `<<Control>>:Withdraw Funds`) to the newly created lifeline is created, as shown in Appendix G, Figure 101. If the transaction type of the sentence is `ResponseToPrimaryActor` or `ResponseToSecondaryActor`, then the subject of the sentence is always “the system” and the object of the sentence is either the primary actor or a secondary actor. Therefore, two sequential messages are generated from the system lifeline to the actor boundary class lifeline and then from it to the actor lifeline.

Rule C2 invokes rules C2.1-C2.4 to process four different types of complex sentences. If the action of the sentence of a condition check sentence (containing keyword `VALIDATES THAT`) is initiated by the system, then a self message (e.g., message 3, Appendix G, Figure 101) is generated at the system lifeline, otherwise a message is

generated from the system lifeline to the actor lifeline (rule C2.1). An instance of the “opt” `CombinedFragment` is generated for the alternative flow branching from the sentence¹⁴. This combined fragment contains an instance of `InteractionUse` which refers to the instance of `Interaction` generated to contain the alternative flow actions (e.g., interaction `Bounded Alternative Flow`, Appendix G, Figure 103). Rule C2.2 processes conditional sentences containing keyword IF-THEN-ELSE-ELSEIF-END. An “alt” `CombinedFragment` is generated and conditions of the sentence (i.e., IF and/or ELSEIF conditions) map to the guards of each operand and the THEN and ELSE actions are mapped to sequence diagram elements in the corresponding operands. Rule C2.3 transforms a parallel sentence into an instance of the “par” `CombinedFragment`. Each sentence connected by keyword MEANWHILE maps to an operand of the “par” combined fragment. For example, step 1 of the specific alternative flow in Table 10 is transformed into an instance of the “par” `CombinedFragment` containing two sets of concurrent messages, as shown in Appendix G, Figure 104. Rule C2.4 transforms an iterative sentence into an instance of the “loop” combined fragment.

Atomic rules C3.1-C3.4 process special sentences containing keywords INCLUDE USE CASE, EXTENDED BY USE CASE, ABORT, and RESUME STEP. For example, step 3 of the specific alternative flow (Table 10) is transformed into a destroy message in Appendix G, Figure 104.

Rule C4 processes global alternative flows, which refer to any step in the reference flow (Table 10). Such a behavior is very hard to model in sequence diagrams: one would need a “opt” combined fragment for each message, which would clutter the diagram. Activity diagrams are a better solution to model this (Chapter 9). Our current solution is to generate an “alt” combined fragment. One of its operands covers all the messages of the reference flow and the rest covers the messages generated for the alternative flow steps. For example, an “alt” combined fragment is generated for the global alternative flow

¹⁴ RUCM requires that an alternative flow branching from the condition check sentence is specified to describe what happens when the validation fails.

(Table 10). Operand [IF NOT: ATM customer enters Cancel] of the combined fragment covers all the elements in the basic flow and operand ELSE contains interaction use Global Alternative Flow, which refers to another interaction covering all the elements corresponding to the steps of the global alternative flow (Appendix G, Figure 102). Also notice that the other two types of alternative flows (i.e., specific and bounded alternative flows) are taken care of by the rules transforming `ConditionCheckSentence` and `ConditionalSentence` complex sentences.

As we will discuss in Section 6.2, transformation rule sets A, B, C and D are applied sequentially. However, while sequence diagrams are being generated, the class diagram already created is further refined by adding more operations. When a message is generated, our approach looks for an operation in the class (the type of the message receiving object) that matches to the description (text) of the part of the sentence (e.g., the predicate). If such an operation does not exist in the class, a new operation is generated, thus refining the class diagram. Also note that by doing so, the class and sequence diagrams are also kept consistent to each other.

Table 13 Summary of rule set C

Rule #	Description
C1	Invoke rules C1.1-C1.6 to process seven (of eight) types of simple sentence.
C1.1 (SV)	InternalTransaction-generate a self-message at the system lifeline.
C1.2 (SVC)	Initiation-generate two sequential messages: actor lifeline (subject) → actor boundary class lifeline → system lifeline (complement). InternalTransaction-generate a self-message for system lifeline. ResponseToPrimaryActor/ResponseToSecondaryActor-generate two sequential messages: system lifeline (subject) → boundary class lifeline of actor (complement) → the actor.
C1.3 (SVCC)	Initiation-generate two sequential messages: actor lifeline (subject) → actor boundary class lifeline → system lifeline (one of the complements). InternalTransaction-generate a self-message for system lifeline. ResponseToPrimaryActor/ResponseToSecondaryActor-generate two sequential messages: system lifeline (subject) → boundary class lifeline of actor (one of the two object complements) → the actor.
C1.4 (SVDO)	Initiation-generate two sequential messages: actor lifeline (subject) → actor boundary class lifeline → system lifeline (direct object). InternalTransaction-generate a message from the system lifeline to the object class lifeline. ResponseToPrimaryActor/ResponseToSecondaryActor-generate two sequential messages: the system lifeline (the subject) → the boundary class lifeline of the actor (the object) → the actor lifeline.
C1.5 (SVDOC)	Initiation-generate two sequential messages: actor lifeline (subject) → actor boundary class lifeline → the system (direct object) lifeline. InternalTransaction-generate a message from the system lifeline to the object class lifeline. ResponseToPrimaryActor/ResponseToSecondaryActor-generate two sequential messages: the system lifeline (the subject) → the boundary class lifeline of the actor (the object or the object complement) → the actor lifeline.
C1.6 (SVIODO SVIODOC)	Initiation-generate two sequential messages: actor lifeline (subject) → actor boundary class lifeline → system lifeline (indirect object). InternalTransaction-generate a message from the system lifeline to the indirect object class lifeline. ResponseToPrimaryActor/ResponseToSecondaryActor-generate two sequential messages: system lifeline (the subject) → boundary class lifeline of the actor (the indirect object) → the actor lifeline.
C2	Invoke rules C2.1-C2.4 to process complex sentence types.
C2.1	ConditionCheckSentence-if the sentence contains an actor to execute the action, generate two sequential messages: the system lifeline (the subject) → the boundary class lifeline of the actor → the actor lifeline. If the sentence executes the action by itself, generate a self message at the system lifeline. Generate a “opt” combined fragment for the alternative flow branching from the sentence and do the following: 1) generate an interaction, 2) generate an interaction use contained in the combined fragment referring to the newly generated interaction, and 3) invoke rules C1-C3 to generate messages for the sentences contained in each alternative flow.
C2.2	ConditionalSentence-generate an “alt” combined fragment and invoke rules C1-C3 to generate messages for the sentences contained in THEN actions and ELSE actions. Conditions of the sentence (e.g., IF condition) map to the conditions of the “alt” combined fragment. If a partial of the sentence (e.g., ELSEIF-THEN) contained in an alternative flow, then a set of messages is generated for the sentences of the alternative flow and contained in the combined fragment.
C2.3	ParallelSentence-generate a “par” combined fragment and invoke rules C1-C3 to generate messages for the sentences connected by the keyword MEANWHILE.
C2.4	IterativeSentence-generate a “loop” combined fragment and invoke rules C1-C3 to generate messages for the sentences in the loop.
C3	Invoke rules C3.1-C3.4 to process each special sentence.
C3.1	IncludeSentence-generate an interaction use referring to the interaction corresponding to the included use case and place the interaction use where the special sentence is invoked.
C3.2	ExtendSentence-generate an interaction use referring to the interaction corresponding to the extending use case and place the interaction use where the special sentence is invoked.
C3.3	AbortSentence-generate a destroy message pointing to the system lifeline.
C3.4	ResumeBackSentence-generate a gate message referring to resume-back-step (message).
C4	Process global alternative flows. The rule also invokes rules C1-C3 to process the steps contained in the global alternative flows.

8.2 Algorithms

Figure 30 presents the high-level algorithm for transforming each use case of a use case model into sequence diagrams contained in the corresponding analysis model.

The `Transform_SD` function has two input parameters: the use case model being transformed into sequence diagrams (instance of metaclass `UseCaseModel` of `UCMeta:ucModel`) and the UML analysis model to contain sequence diagrams to be generated (instance of metaclass `Model` of the UML 2 metamodel). The function outputs the refined UML analysis model (`umlModel`) containing the generated sequence diagrams for each use case of the use case model. It starts by iterating over each use case of the use case model to `invoke_rule_C0()` to generate sequence diagrams for each use case. The basic idea of lines 1-5 is to process included or extending use cases of a use case beforehand. By doing so the including and extended use case can refer to these included or extending use cases through instances of `InteractionUse`; therefore the efficiency of the algorithm is improved. Operation `invoke_rule_C0()` corresponds to the composition rule C0 in Table 13. It starts by processing the global alternative flow(s) of a use case by `invoke_rule_C4` (lines 6-8). Then the operation iterates over each step of the basic flow of the use case and invokes different rules according to the types of the step sentences (lines 9-13). If the sentence is a simple (complex, or special) sentence then rule C1 (C2 or C3) is invoked. Operations `invoke_rule_C1()`, `invoke_rule_C2()`, and `invoke_rule_C3()` further invoke rule sets C1.1-1.6, C2.1-2.4, and C3.1-3.4, respectively, to process different types of simple, complex and special sentences.

Algorithm Transform_SD (ucModel:ucm::UseCaseModel, umlModel:uml::Model)
Input ucModel : ucm::UseCaseModel --- The use case model being transformed into sequence diagrams
Output umlModel : uml::Model --- The refined UML model containing generated sequence diagrams
Begin
1. ucModel.modelElements.select{e | ucm::UseCase.isInstanceOf(e)}.each{uc |
2. **if** uc.include.size >0 **then** uc.include.forAll{u | invoke_rule_0(u.addition, umlModel)}
3. **else if** uc.extend.size >0 **then** uc.extend.each{u | invoke_rule_0(u.extendedCases, umlModel)}
4. **else** invoke_rule_C0(uc, umlModel) **end**
5. }
End

Operation invoke_rule_C0(uc:ucm::UseCase, umlModel:uml::Model)
Declare: basicFlow : ucm::BasicFlow --- The basic flow of the use case
 altFlows : Sequence(ucm::AlternativeFlow) --- The alternative flows of the use case
 bf_steps : OrderedSet(ucm::Sentence) --- The steps of the basic flow of the use case
Begin:
6. altFlows.each{alt|
7. **if** alt.isInstanceOf(ucm::GlobalAlternative) **then** invoke_rule_C4(s, interaction, uc)
end
8. }
9. bf_steps.each{s|
10. **if** s.isInstanceOf(ucm::SimpleSentence) **then** invoke_rule_C1(s, interaction, uc) **end**
11. **if** s.isInstanceOf(ucm::ComplexSentence) **then** invoke_rule_C2(s, interaction, uc) **end**
12. **if** s.isInstanceOf(ucm::SpecialSentence) **then** invoke_rule_C3(s, interaction, uc) **end**
13. }
End

Figure 30 Transformation algorithm for generating sequence diagrams

CHAPTER 9 GENERATE ACTIVITY DIAGRAMS

Use case modeling, through use case diagrams and use case textual specifications, is commonly applied to structure and document requirements (e.g., [58]). In this context, UML Activity diagrams are often used to: 1) Visualize use case scenarios to better understand and analyze them (e.g., [85]), which becomes paramount when use cases are large and complex; 2) Model business processes, work and data flows, of which information is embedded in use case descriptions (e.g., [15]); 3) Complement analysis models by providing an additional, complementary view to class and interaction diagrams (e.g., [33]), and 4) Generate test cases complying with use cases (e.g., [71]). Automated support to transform a use case description into an (initial) activity diagram is therefore important.

In this section, we focus on the RUCM to activity diagrams transformation of aToucan. Specifically, aToucan can automatically generate two types of activity diagrams for each use case: A *detailed activity diagram* shows the main use case flow as well as all alternative flows in one activity diagram; An *overview activity diagram*, on the other hand, only details the main use case flow while the alternative flows are detailed in parts of the sequence diagram aToucan generates for the use case. The activity diagram of the main flow refers to parts of the sequence diagram thanks to the UML 2.0 notions of `CallBehaviorAction` and `Interaction`. Overview activity diagrams therefore help to handle complexity in use case descriptions (complex flows, numerous flows). Our approach can also automatically attach data flow information to generated (overview or detailed) activity diagrams. This is useful to measure complexity or facilitate data flow-based testing for instance [105].

The rest of the section is organized as follows. We present an overview of the approach for the generation of activity diagrams in Section 9.1 and then detail transformation rules (Section 9.2), transformation algorithm (Section 9.3) and traceability (Section 9.4).

9.1 Overview

In this section, we use Figure 31 as a running example. It shows a piece of the use case description of Table 10: Figure 31 (a). The first transformation is to automatically transform a textual UCMOD (Figure 31 (a)) into an instance of UCMeta (Figure 31 (b)) through a set of transformation rules. For example, basic flow step 8 of Figure 31 (a) is transformed into an instance of `ConditioncheckSentence` (Figure 31 (b)). Notice in Figure 31 (b) that this `ConditioncheckSentence` instance is linked to a `BasicFlow` instance (step 8 is part of the basic flow in Table 10) of the `UseCaseSpecification` of `UseCase Withdraw Funds`. Figure 31 (b) does not show how the sentence of step 8 is further transformed into instances of UCMeta (e.g., verb, subject).

Second, the UCMeta instance is automatically transformed into a UML analysis model through another set of transformation rules. The (UML 2.0) analysis model contains a class diagram, and a sequence and an activity diagram for each use case. Generating class and sequence diagrams is discussed in Chapter 7 and Chapter 8, respectively. In this section, we particularly focus on the transformation to activity diagrams.

As mentioned earlier, two types of activity diagrams can be generated for a use case, i.e., from an instance of UCMeta. A *detailed activity diagram* shows the main use case flow as well as all alternative flows in one activity diagram: Figure 31 (d); whereas an *overview activity diagram* only details the main use case flow while the alternative flows are detailed in parts of the sequence diagram generated for the use case (instances of `Interaction`): Figure 31 (c). To illustrate the difference, first note that the parts of Figure 31 (c) and (d) highlighted with rectangles detail the main flow of the use case in the same way: one can recognize step 8 (“The system VALIDATES THAT ...”) followed by a decision node (the validation may be successful or not). In the detailed activity diagram (Figure 31 (d)) the alternative flow (i.e., when the validation fails) is specified in its entirety (circled set of nodes). Instead, in the overview activity diagram (Figure 31 (c)), the alternative flow leads to a node labeled `ref INTERACTION ...`, specifying that the

alternative flow can be obtained from an interaction, specifically from a part of the sequence diagram aToucan generated for the use case.

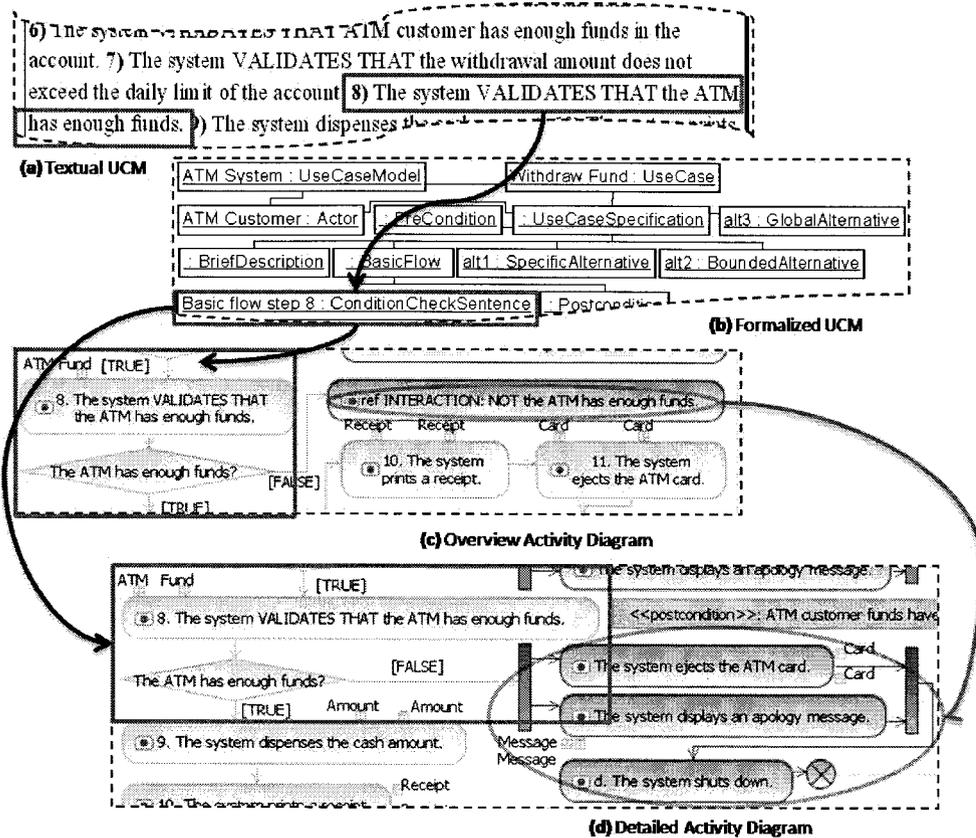


Figure 31 Running example – activity diagram generation

Overview activity diagrams are similar to UML Interaction Overview Diagrams [75] in the sense that both use `CallBehaviorAction` referring to instances of `Interactions` of sequence diagrams. However, UML Interaction Overview Diagrams only “focus on the overview of the flow of control where the nodes are `Interactions` or `InteractionUses`” [75] while our Overview Activity Diagram can contain other activity nodes such as instances of `CallOperationAction`.

During the transformations, two sets of traceability links are established: between the elements in the textual UCM and the elements of the instance of UCMeta, and between these instances and the model elements of the UML analysis model. For instance, step 8

of the use case specification of Table 10 is linked to the `ConditionCheckSentence` instance highlighted in Figure 31 (b), which is itself linked to the activity nodes labeled 8 (*The System ...*) in Figure 31 (c) and (d). Notice that when necessary, direct traceability links between the textual UCMOD and the analysis model can be easily derived from the transitive closure of these two sets of traceability links.

9.2 Transformation Rules

The transformation from an instance of `UCMeta` to activity diagrams involves 19 rules, summarized in Table 14. Subscripts on rule numbers (Column 1, Table 14) indicate the type of the rule: "c" and "a" denote composite and atomic rules, respectively; a composite rule is decomposed whereas an atomic rule is not.

Rule D1 invokes rules D1.1-D1.4 to generate an activity diagram for each use case. Rules D1.1-D1.3 process three types of sentences: `SimpleSentence`, `ComplexSentence`, and `SpecialSentence`. Rule D1.4 processes the `GlobalAlternativeFlows` of a use case. Rules D1.5 and D1.6 transform the precondition and the postcondition of a use case into instances of `Constraint` (a metaclass of the UML 2.0 metamodel) attached to the generated activity (precondition) and corresponding `FlowFinalNode` (postcondition). Atomic rules D1.2.1-D1.2.4 are invoked by composite rule D1.2 to process four different types of complex sentences that lead to different control flows in the activity diagram (e.g., decision node in rule D1.2.1). Atomic rules D1.3.1-D1.3.4 are invoked by rule D1.3 to process four different types of special sentences: to specify include and extend relations between use cases, to specify abort and resume. Rule D1.4 transforms a global alternative flow. Recall that a global alternative flow refers to any step in the reference flow (Section 3.1.1). For example, the global alternative flow of use case *Withdraw Funds* (Table 10) refers to every step of the basic flow; the ATM customer can cancel the transaction at any time of the execution of the use case. To model this in an activity diagram, we transform the flow into an instance of `AcceptEventAction`, `InterruptibleActivityRegion` and a set of actions corresponding to the steps of that flow. `AcceptEventAction` is an action that waits for the occurrence of an event meeting

a specified condition" [75] and `InterruptibleActivityRegion` (e.g., the basic flow of use case *Withdraw Funds*) is used to abort all flows in the region when an `AcceptEventAction` (e.g., ATM customer enters Cancel—the condition of the global alternative flow) occurs. Thanks to this modeling feature of UML 2.0, we can easily model global alternative flows.

Note that rules D1.2.1, D1.2.2, and D1.4 generate different sets of model elements for detailed and overview activity diagrams. For example, if an overview activity diagram is generated, composite rule D1.2.1 generates an instance of `CallBehaviorAction` to refer to the `Interaction` corresponding to the alternative flow of a condition check sentence; otherwise rule D1.2.1 invokes rules D1.1-D1.3 to process the sentences of the alternative flow to generate actions, edges, etc, and create a detailed activity diagram.

Table 14 Summary of rule set D

Rule #	Description
D1 _c	Generate an activity diagram for a use case.
D1.1 _a	Generate an instance of CallOperationAction for each simple sentence.
D1.2 _c	Invoke rules D1.2.1-D1.2.4 to process each complex sentence.
D1.2.1 _c	ConditionCheckSentence: Generate a CallOperationAction and a DecisionNode. Invoke rules D1.1-D1.3 to handle the sentences contained in the alternative flow corresponding to the sentence (detailed activity diagram) or refer to the Interaction corresponding to the alt. flow (overview activity diagram).
D1.2.2 _c	ConditionalSentence: Generate a DecisionNode. Invoke rules D1.1-D1.3 to process sentences contained in the sentence and its alt. flow (detailed activity diagram) or refer to the Interaction corresponding to the alt. flow (overview activity diagram) if such an alt. flow exists.
D1.2.3 _c	ParallelSentence: Generate a ForkNode and a JoinNode. Invoke rules D1.1-D1.3 to process the concurrent sentences contained the parallel sentence.
D1.2.4 _c	IterativeSentence: Generate a DecisionNode. Invoke rules D1.1-D1.3 to process sentences contained in the iterative sentence.
D1.3 _c	Invoke rules D1.3.1-D1.3.4 to process each special sentence.
D1.3.1 _a	IncludeSentence: Generate a CallBehaviorAction that refers to the Interaction corresponding to the included use case.
D1.3.2 _a	ExcludeSentence: Generate a CallBehaviorAction that refers to the Interaction corresponding to the extending use case.
D1.3.3 _a	AbortSentence: Generate a FlowFinalNode.
D1.3.4 _a	ResumeStepSentence: Generate a ControlFlow edge back to the node corresponding to the step specified in the ResumeStepSentence.
D1.4 _c	GlobalAlternativeFlow: Generate an AcceptEventAction and InterruptibleActivityRegion. Invoke rules D1.1-D1.3 to process the sentences of the alt. flow (detailed activity diagram) or refer to the Interaction corresponding to the alt. flow (overview activity diagram).
D1.5 _a	Precondition: Generate a Constraint as the precondition of the activity. The content of the constraint is the precondition of the use case.
D1.6 _a	PostCondition: Generate a Constraint for each flow final node as its postcondition. The content of each constraint corresponds to the postcondition of each flow of events of the UCMOD.
D2 _c	Attach data flow information to an activity diagram.
D2.1 _a	SimpleSentence with transaction type Initiation, ResponseToPrimaryActor or ResponseToSecondaryActor: Generate an OutputPin for the CallOperationAction generated for the NPs of the sentence.
D2.2 _a	SimpleSentence with transaction type InternalTransaction: Generate an InputPin and an OutputPin for the CallOperationAction generated for the sentence (excluding "the system" and actors).
D2.3 _a	ConditionCheckSentence: Generate an InputPin for the CallOperationAction generated for the NPs of the sentence (excluding "the system" and actors).

Rule 2 invokes rules D2.1-D2.3 to attach data flow information to an already generated activity diagram. These rules generate instances of either InputPin or OutputPin for

each call operation action. These input and output pins correspond to entity classes that have been generated from the NPs contained in use case sentences when the class diagram of the system was generated (Chapter 7). For example, the basic flow step 8 of use case *Withdraw Funds* (Table 10) is a condition check sentence (Figure 31 (a)). It is transformed into an action and a decision node by rule D1.2.1: (Figure 31 (c) and (d)). Because the NP "enough funds"—i.e., the object of the condition (simple sentence) of the condition check sentence at step 8—has been transformed into class `Fund` when the class diagram was generated, we attach an instance of `InputPin` (`pin`) to the action corresponding to step 8 and type the pin with class `Fund`: `pin.type = Fund`, as show in Figure 31 (c) and (d).

The rationale for adding data flow to an activity diagram is the following. Steps of a UCS can be one of the following five types: 1) `Initiation`: the primary actor sends a request and data to the system; 2) `Validation`: the system validates a request and data; 3) `InternalTransaction`: the system alters its internal state (e.g., recording or modifying something); 4) `ResponseToPrimaryActor`: the system replies to the primary actor with a result; 5) `ResponseToSecondaryActor`: the system sends requests to a secondary actor. We generate data flow through input and output pins according to these definitions as follows: We generate output pins for actions in the activity diagram that correspond to use case steps of type `Initiation`, `ResponseToPrimaryActor`, and `ResponseToSecondaryActor` since these sentences either output data (`Initiation`) or send a result to actors (`ResponseToPrimaryActor` or `ResponseToSecondaryActor`); Since use case steps of type `InternalTransaction` specify that the system records or modifies data, we generate input and output pins for the actions corresponding to these sentences; Since condition check sentences are all of type `Validation` and they validate a request and data, input pins should be generated for the corresponding actions in the activity diagram. As suggested earlier, the pins are typed by the entity classes that compose the domain model (class diagram) automatically created from use case descriptions by aToucan (Chapter 7), which are identified by analyzing sentences (e.g., subjects).

Notice that each rule is further specified by a precondition, although these preconditions are not shown in Table 14. As a simple example, the precondition of rule D1.2.3 specifies that the sentence being transformed into model elements is a parallel sentence (Section 3.2) and that an activity has been generated for the use case and is available to contain the elements being generated by the rule (Figure 32).

As an example, consider composite rule D1.2.3 (Figure 32) where a parallel sentence is transformed into a fork node and a join node and the concurrent sentences are transformed into actions between the fork and join nodes by invoking rules D1.1-D1.3. Two traceability links are established between the fork and join nodes and the parallel complex sentence during this transformation. Traceability links are also established between each concurrent sentence (sentences connected by keyword MEANWHILE in the parallel sentence) and their corresponding actions when rules D1.1-D1.3 are executed. The specification of the rule is shown as the Kermeta language [55], which is a metamodeling language similar to the Object Constraint Language (OCL) [76] in Figure 32. Some of statements are annotated with *italic fonts* starting with “---”.

```

operation transform_rule_1.2.3(uc : ucm::UseCase, sen : uml::ParallelSentence, activity : uml::Activity,
    partitions : Set<uml::ActivityPartition>, nodes : Sequence<uml::ActivityNode>,
    edges : Sequence<uml::ActivityEdge>) : Void
pre precondition is do
    sen.isInstanceOf(ucm::ParallelSentence) and activity != void
end
is do
    var message : traceability::Message ---Instance of Message of the traceability metamodel
    message := trace_helper.createMessage("1.2.3", "ConditionalSentence -> ForkNode + JoinNode")
    var sens : Sequence<Sentence> init Sequence<Sentence>.new
    sens.addAll(sen.parallelActions) ---The concurrent sentences contained in the parallel sentence
    var forkNode : uml::ForkNode init uml::ForkNode.new
    var joinNode : uml::JoinNode init uml::JoinNode.new
    var inEdge : uml::ActivityEdge ---The incoming edge to the fork node
    inEdge := edges.last() ---The source of the incoming edge is a previously generated activity node.
    inEdge.target := forkNode ---The target of the incoming edge is the newly generated fork node.
    nodes.add(forkNode) --- Store the fork node to the node collection.
    nodes.add(joinNode) --- Store the join node to the node collection.
    activity.node.add(forkNode) --- Connect the fork node to the activity.
    activity.node.add(joinNode) --- Connect the join node to the activity.
    forkNode.inPartition.add(inEdge.source.inPartition.one) ---Add the fork and join nodes to-
    joinNode.inPartition.add(inEdge.source.inPartition.one) ---the partition as its previous node.
    sens.each {s | --- Process each concurrent sentence
        var outEdge : uml::ActivityEdge init uml::ControlFlow.new
        outEdge.source := forkNode --- The outgoing edge from the fork node
        var parNodes : Sequence<ActivityNode> init Sequence<ActivityNode>.new
        var parEdges : Sequence<ActivityEdge> init Sequence<ActivityEdge>.new
        parEdges.add(outEdge) --- Maintain a branch for each concurrent sentence
        parNodes.add(forkNode)
        activity.edge.add(outEdge)
        if s.isInstanceOf(ucm::SimpleSentence) then
            transform_E1_1(uc, s.asType(ucm::SimpleSentence), activity, partitions, parNodes, parEdges)
        end
        if s.isInstanceOf(ComplexSentence) then
            transform_E1_2(uc, s.asType(ucm::ComplexSentence), activity, partitions, parNodes, parEdges)
        end
        if s.isInstanceOf(SpecialSentence) then
            transform_E1_3(uc, s.asType(ucm::SpecialSentence), activity, partitions, parNodes, parEdges)
        end
        var toJoinEdge : uml::ActivityEdge init uml::ControlFlow.new
        toJoinEdge.source := parNodes.last --- The edge connecting to the join node
        toJoinEdge.target := joinNode
        parEdges.add(toJoinEdge)
        activity.edge.add(toJoinEdge)
    }
    var outgoingEdge : uml::ActivityEdge init uml::ControlFlow.new
    outgoingEdge.source := joinNode --- The outgoing edge from the join node
    edges.add(outgoingEdge)
    activity.edge.add(outgoingEdge)
    --- Establish traceability links
    if forkNode != void then trace_helper.createTLink(sen, forkNode, message) end
    if joinNode != void then trace_helper.createTLink(sen, joinNode, message) end
end

```

Figure 32 Transformation rule D1.2.3 in the Kermeta language [55]

9.3 Algorithm

Rules are structured in a hierarchy, as illustrated by the numbering in Table 14, which indicates that some rules have to be executed one after the other while others can be executed independently. This facilitates the modification, addition, and deletion of rules and also simplifies the algorithm to apply them. The invocation of each rule is determined by its precondition.

Figure 33 presents the high level algorithm for transforming an instance of UCMeta into an activity diagram (an instance of the UML 2.0 activity metamodel). The algorithm is formalized using pseudocode where some variables are typed as model elements in UCMeta (prefixed with *ucm::*) and the UML activity diagram metamodel (prefixed with *uml::*), which are highlighted in `courier new` font.

The `Transform` function has one input parameter: the use case being transformed into an activity (instance of metaclass `UseCase` of UCMeta: `uc`) and outputs the generated activity (`activity`) for the use case. It starts by generating an activity, its initial node, and its partitions (lines 1-7): partitions (or swimlanes) are created for the system and each actor interacting with the use case. Second, the step sentences of the basic flow of the use case are transformed by invoking rules D1.1-D1.3 (lines 8-12). Which rules to invoke depends on the type of a sentence. Third, rule D1.4 is invoked to process the global alternative flows of the use case (line 13). Fourth, the final node, precondition, and postcondition of the activity are generated (lines 14-22). Fifth, rule D2 is invoked to generate data flow information for the already generated activity diagram (line 23). Finally, a traceability link is generated between the use case and the activity (line 24), as described next. Note that more specific links between the use case elements (e.g., sentence) and model elements of the activity are generated in each invoked rule (i.e., rules D1.1-D1.4 and D2).

Algorithm Transform (uc) : activity

Input uc : ucm::UseCase --- The use case being transformed into an activity
Output activity : uml::Activity --- The activity transformed from the use case
Declare basicFlowSteps : Sequence(ucm::Sentence) --- The basic flow steps of the use case
altFlows : Set(ucm::AlternativeFlow) --- The alternative flows of the use case
initialNode : uml::InitialNode --- The initial node of the activity
partitions : Set(uml::ActivityPartition) --- The partitions of the activity
finalNode : uml::ActivityFinalNode --- The final node of the activity
precondition : uml::Constraint --- The precondition of the activity
postcondition : uml::Constraint --- The postcondition of the main branch of the activity
tlink : traceability::Trace --- The traceability link between the use case and the activity
msg : traceability::Message--- The tlink message describing the transformation.

Begin

```

1. activity ← uml::Activity.new
2. activity.name ← uc.name
3. initialNode ← uml::InitialNode.new
4. activity.node.add(initialNode)
5. partitions.add(CreatePartitionForSystem(uc))
6. partitions.add(CreatePasrtitionForEachActor(uc))
7. activity.partition.addAll(partitions)
8. basicFlowSteps.each { s : ucm::Sentence |
9.   if (s.isInstanceOf(ucm::SimpleSentence) then invoke_rule_1.1(uc, s, activity) end
10.  if (s.isInstanceOf(ucm::ComplexSentence) then invoke_rule_1.2(uc, s, activity) end
11.  if (s.isInstanceOf(ucm::SpecialSentence) then invoke_rule_1.3(uc, s, activity) end
12. }
13. invoke_rule_1.4(uc, altFlows.select
      {a|a.isInstanceOf(ucm::GlobalAlternativeFlow)}, activity)
14. finalNode ← uml::ActivityFinalNode.new
15. activity.node.add(finalNode)
16. precondition ← uml::Constraint.new
17. precondition.name ← uc.specification.preCondition.content
18. activity.ownedRule.add(precondition)
19. postcondition ← uml::Constraint.new
20. postcondition.name ← uc.specification.flows.
21.   select {f|f.isInstanceOf(ucm::BasicFlow)}.one.postCondition.content
22. activity.ownedRule.add(postcondition)
23. invoke_rule_2(uc, activity)
24. tlink ← createTLink(uc, activity, message)
25. return activity

```

End**Figure 33 Transformation algorithm for generating activity diagrams****9.4 Traceability**

We establish two sets of traceability links during the transformation from a textual UCMOD to activity diagrams: from the UCMOD to the instance of UCMeta; from the UCMeta instance to the automatically generated activity diagrams. If necessary, direct

traceability links from the textual UCMOD to the activity diagrams can be derived from these two sets.

Traceability links from UCMOD to UCMeta link the fields of the use case template used to document textual UCSs to instances of the corresponding metaclasses in UCMeta. For example, field *Brief Description* of the use case template is linked to an instance of metaclass `BriefDescription` of UCMeta. A sentence in the brief description is then linked to an instance of metaclass `Sentence` of UCMeta. For example, as shown in Figure 31 (a) and (b), the basic flow step 8 of use case *Withdraw Funds* is transformed into `Basic flow step 8 : ConditionCheckSentence` of the intermediate model while a traceability link is established between these two elements. We believe that it is not cost effective to establish links at a finer granularity (e.g., between elements of sentences and UCMeta metaclass instances).

Regarding the second set of traceability links, UCMeta metaclass instances are linked to corresponding model elements in the UML activity metamodel based on our transformation rules. For example, we establish two traceability links between a condition check sentence (e.g., `Basic flow step 8 : ConditionCheckSentence` of the UCMeta instance as shown in Figure 31 (b)) and its corresponding action (an instance of `CallOperationAction`, e.g., `action 8. The system VALIDATES THAT the ATM has enough fund` as shown in Figure 31 (c)) and decision node (e.g., `The ATM has enough fund` as shown in Figure 31 (c)) generated during the transformation when rule D1.2.1 is invoked, respectively.

CHAPTER 10 AUTOMATION (ATOUCAN)

The main objective of our approach is to support the transition from textual requirements to (initial) UML class and sequence diagrams. We designed our tool (aToucan) by following a number of standard principles, such as modularity, extensibility, and modifiability. Our main goal was to obtain a tool that could be easily extended and would easily accommodate certain types of changes.

Transformation rules, either for FormalizeUCM or GenerateUML, are structured into packages, e.g., we have a package for the template rule set (part of FormalizeUCM, as discussed in Chapter 5) and for rule set A (part of GenerateUML, as discussed in Section 6.1). This allows us to have well-structured sets of rules thus facilitating their modifications and extensions. Indeed, even though we aim to define sets of transformation rules that are as complete as possible, it is expected that they will evolve and be refined over time.

aToucan relies on a number of existing technologies.

- 1) aToucan is built as an Eclipse plug-in, using the Eclipse development platform.
- 2) UCMeta is implemented as an Ecore model, using Eclipse EMF [27]. Eclipse EMF is a modeling and code generation engine for building applications based on a structured data model (e.g., UCMeta). In our case, Eclipse plug-ins are generated.
- 3) We use the Stanford Parser [101] as a NL parser. It is written in Java and generates (1) a syntactic parse tree for a sentence and (2) the sentence's grammatical dependencies (e.g., *subject*, *direct object*). Though we chose the Stanford Parser, our design is not necessary bound to any special NL parser, thanks to the use of separate packages (e.g., for the parser and transformations) and our use of the adapter design pattern (to adapt to different parsers).

- 4) The generation of the UML analysis model relies on Kermeta [55]. Kermeta is a metamodeling language, also built on top of the Eclipse platform and EMF, which also supports model transformations. One interesting aspect of Kermeta transformation mechanism is that a transformation can manipulate both the source and target model elements. Kermeta also supports packages, inheritance, classes, and operations so that transformation rules can be well organized. In addition, design-by-contract principles can be followed when specifying transformation operations: Kermeta supports operation pre and post conditions and class invariants.
- 5) The target UML analysis model is instantiated using the Eclipse UML2 project, which is an EMF-Based implementation of the UML 2 standard, and our Analysis Profile Model that adds class stereotypes <<Control>>, <<Entity>>, and <<Boundary>>.

10.1 FormalizeUCM subsystem

FormalizeUCM is implemented as one of the two subsystems of the aToucan tool. As shown in Figure 34, Transformation Engine 1) takes a textual UCMOD as input (Textual UCM package), 2) triggers Template Parser (through Template Parser Interface) and The Stanford Parser (through NL Parser Interface and NL Parser Adapter), 3) invokes the transformation rules (Transformation Rules), 4) generates Formalized UCM, which is an instance of the UCM Ecore Metamodel (completed by Constructing UCM engine), and 5) establishes Traceability Links (Establishing TLink Engine).

As shown in Figure 34, if a different NL parser is used, we just need to change the NL Parser Adapter connecting to the specific NL parser. The Transformation Rules package is composed of a set of packages, which correspond to the different transformation rule sets described in Chapter 5.

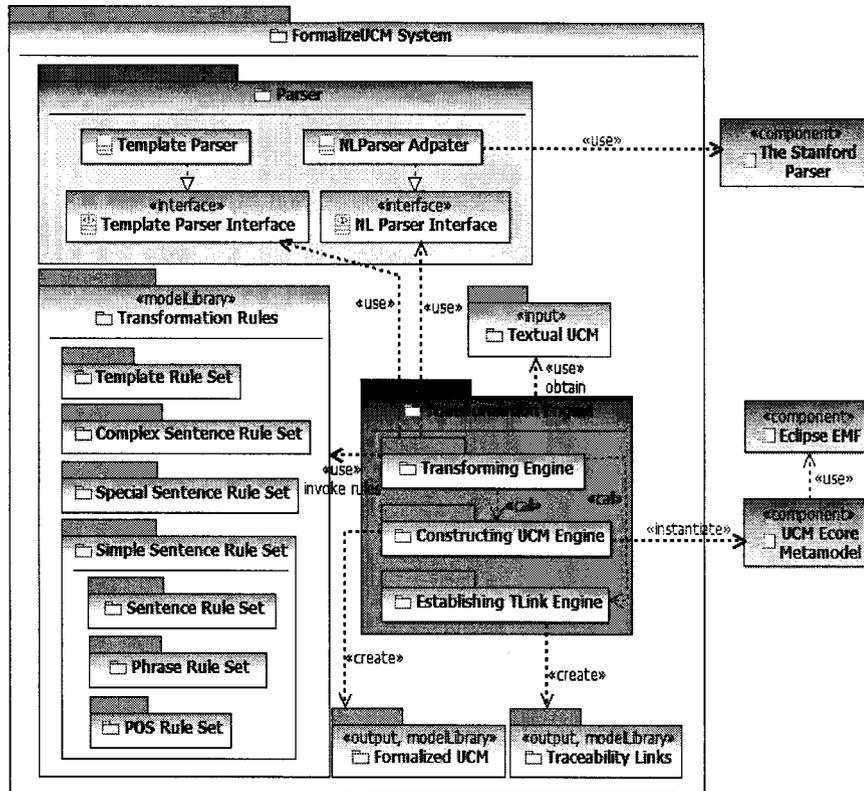


Figure 34 Architecture of the FormalizeUCM subsystem

10.2 GenerateUML subsystem

GenerateUML is also implemented as a subsystem of the aToucan tool. As shown in Figure 35, Transformation Engine 1) takes a Formalized UCM as input, 2) invokes the Transformation Rules, 3) generates Analysis UML Model, which is an instance of UML2 and Analysis Profile Model, and 4) establishes Traceability Links. The generated analysis model and traceability links are queried by transformation rules and the transformation engine through Query Trace Model Helper and Query Target Model Helper to facilitate subsequent transformations. During transformations, the source model must be queried as well (Query Source Model Helper).

The Transformation Rules package is composed of a set of packages, which correspond to the different transformation rule sets described in Chapter 6 (Rule Set A), Chapter 7 (Rule Set B), Chapter 8 (Rule Set C), and Chapter 9 (Rule Set D).

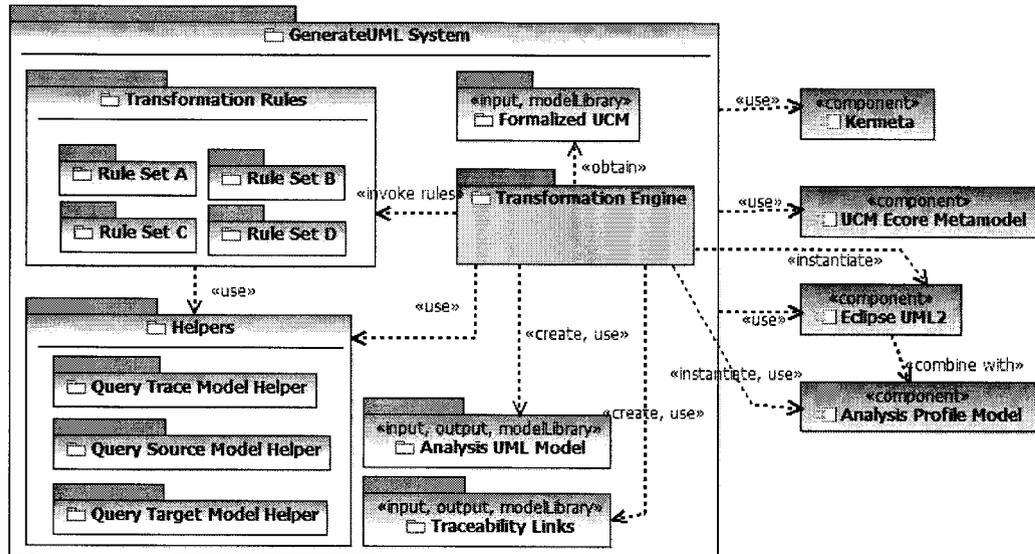


Figure 35 Architecture of the GenerateUML subsystem

CHAPTER 11 EVALUATION FRAMEWORK

Recall that the overall objective of aToucan is to automatically generate a UML analysis model from a textual RUCM model. Our evaluation frameworks consist of a series of empirical studies. This framework, as shown in Figure 36, has four objectives:

- G1: evaluating RUCM with respect to its applicability (Chapter 12);
- G2: evaluating RUCM with respect to its impact on the quality of manually derived analysis models (Chapter 12);
- G3: evaluating our transformation method with respect to its effectiveness by comparing our tool's automatically-generated class and sequence diagrams with those manually created by 4th year undergraduate students (Chapter 13 and Chapter 14);
- G4: evaluating our automated transformation method by analyzing the similarity of the aToucan generated models when compared to reference class and sequence diagrams designed by experts (Chapter 13 and Chapter 14)

Five case studies have been performed to evaluate activity diagrams generated by aToucan. The detailed discussion on the evaluation is provided in Chapter 15. Two industrial case studies have been performed to evaluate the automated generation of class and sequence diagrams (Chapter 16).

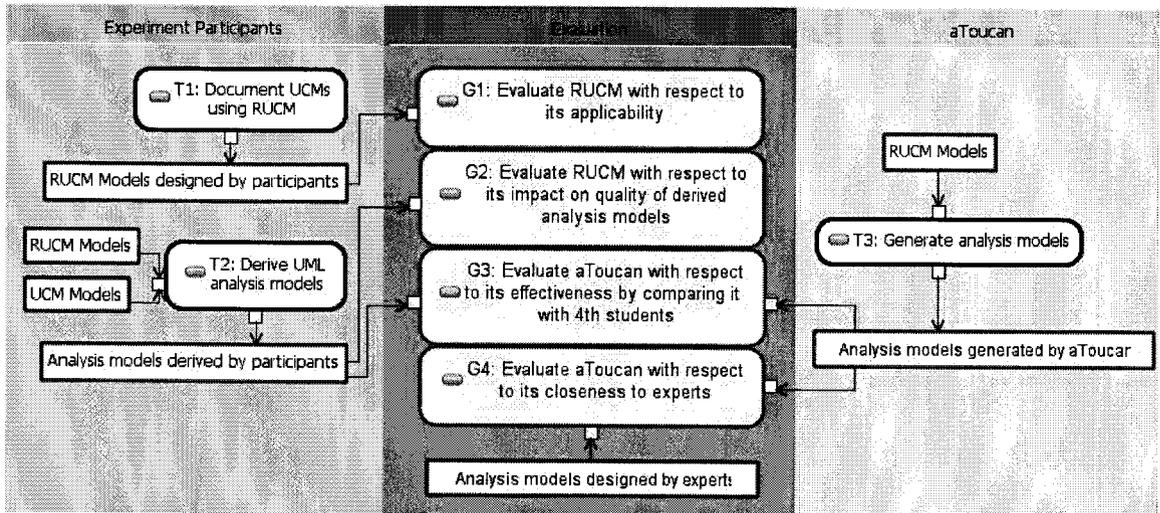


Figure 36 Evaluation framework

CHAPTER 12 EVALUATION OF RUCM

RUCM is composed of a set of well-defined restriction rules and a modified use case template (Chapter 3). The goal is two-fold: (1) restrict the way users can document use case specifications in order to reduce ambiguity and (2) facilitate automated analysis in order to provide tool support to derive initial analysis models, which in UML are typically composed of class diagrams, interaction diagrams, and possibly other types of diagrams and constraints. The derivation of analysis models from UCMods is currently poorly supported by tools and the process is mostly manual.

Though the proposed restriction rules and template are based on a clear rationale, two main questions need to be investigated. Do users find them too restrictive or impractical in certain situations? Second, do the rules and template have a positive, significant impact on the quality of the constructed analysis models? To investigate these questions, we performed and report on controlled experiments, which evaluate the restriction rules and use case template in terms of whether they are easy to apply while developing UCMods and facilitate understanding of UCSs, and whether they help obtain higher quality analysis models in terms of correctness, completeness, and redundancy.

The rest of the section is organized as follows. The related work is reported in Section 12.1. The experimental evaluation of RUCM is presented in Section 12.2 (experiment planning), Section 12.3 (experiment results and analysis), and Section 12.4 (threats to validity). Last we conclude in Section 12.5.

12.1 Related work

Some empirical studies have been conducted to evaluate the impact of applying restriction rules on the quality of UCSs. Achour et al. [4] investigated the effectiveness of the CREWS rules [4] in terms of the completeness and structuredness of UCSs: UCSs were evaluated by comparing them to “expert” UCSs developed by the authors of the paper. The experiment results show that the application of the rules produced more

complete and better structured UCSs. Phalp et al. [83] conducted an empirical study to compare two sets of writing rules: the CREWS rules and CR rules [22] (leaner than the CREWS rules). The overall quality of UCSs was evaluated based on seven quality factors, referred as the ‘7Cs of communicability’ [82]: coverage (a use case should contain all the required information), cogent (a sentence should follow a logical path), coherent (sentences should be all connected by, for example, repeating a noun), consistent abstraction, consistent structure, consistent grammar, and consideration of alternatives. The experiment results from [83] show that the leaner set of rules (the CR rules) results in less learning overhead than the CREWS rules and performs at least as well as the CREWS rules, in terms of the overall quality of produced UCSs. Anda et al. [7] conducted a similar experiment to compare three different sets of guidelines: minimal guidelines (guidelines on identifying actors and use cases), template guidelines (a commonly applied actor and use case template based on templates proposed in [21, 59, 90]), style guidelines (modified version of the CREWS rules, focusing on the documentation of the flow of events of each use case). UCSs were evaluated in terms of their understandability, usefulness, and quality. The experiment results show that the template guidelines led to the highest understandability, usefulness, and overall quality, and that the style guidelines performed better than the minimal guidelines. The authors also suggest that combining the style guidelines with the template guidelines might further improve quality attributes of UCSs to compare with independently applying the template or style guidelines. This is exactly the case of our RUCM.

12.2 Experiment Planning

In this section, we follow the experiment reporting template proposed in [108]. All aspects of the experiments we conducted to assess our use case template and restriction rules are described and justified.

12.2.1 Experiment definition

We are interested in the applicability of the restriction rules, combined with the use case template we proposed. We refer to a use case model with UCSs that follow our restriction rules and template as a *restricted* use case model. We are also interested in the impact of a restricted use case model on the quality of analysis models that are manually derived from it, for instance by following standard guidelines for building analysis models (e.g., [14]). Indeed, if the restriction rules actually reduce ambiguity, then such models should exhibit higher quality.

The experiment objectives are formulated as Goal-Question-Metric (GQM) goals [9] as shown in Table 15 and Table 16. The evaluation of Goal 1 is a necessary pre-requisite to the investigation of Goal 2: we need to ensure that the restriction rules can be applied at a reasonable level of correctness. If the result of the experiment for Goal 1 shows that the restriction rules are applicable, then we can go further and investigate whether these RUCM has an impact on the quality of manually generated analysis models (class and sequence diagrams in our experiment).

Table 15 Goal 1

<i>Analyze</i>	the restriction rules
<i>for the purpose of</i>	characterizing
<i>with respect to</i>	the applicability of each restriction rule
<i>from the point of view of</i>	the requirements engineer
<i>in the context of</i>	4 th year undergraduate students defining use case models

Table 16 Goal 2

<i>Analyze</i>	the restriction rules and the use case template of RUCM
<i>for the purpose of</i>	Evaluating
<i>with respect to</i>	their impact on quality of derived analysis models
<i>from the point of view of</i>	the system analyst
<i>in the context of</i>	4 th year undergraduate students manually deriving analysis models from use case models

Regarding Goal 1, the applicability of each restriction rule is characterized in terms of *Error Rate*, *Understandability*, *Applicability*, and *Restrictiveness*. The experiment for Goal 2 has one independent variable, namely *Method*, with two treatments corresponding to the usage or not of our restriction rules and proposed template. The alternative was to

use one of the well-known use case templates and no restriction rules in the textual description of UCSs. The experiment for Goal 2 has four dependent variables, namely the quality (*Correctness*, *Completeness*, and *Redundancy*) of analysis class diagrams, the quality (*Correctness*, *Completeness*, *Redundancy*, and *Consistency* to the Boundary-Control-Entity principle [15]) of analysis sequence diagrams, the consistency between analysis class and sequence diagrams, and the correctness of responses to a comprehension questionnaire designed for this experiment.

12.2.2 Context selection and subjects

The context of both experiments is a fourth year Software Engineering course at Carleton University, Ottawa, Canada. Since this course is the one-but-last software engineering course in the curriculum, it is an opportunity for the students to use the skills they have acquired in at least three previous courses regarding requirements elicitation and object-oriented analysis and design. The subjects selected were 34 (the first experiment) and 39 (the second experiment) students registered in the course in 2008 and 2009, respectively.

A presentation was first provided to the students before each experiment, focusing on the restriction rules and the use case template, and how they fit into the requirements elicitation and specification phase. An assignment was designed to train the students on how to apply the restriction rules and the use case template before the experiments. The result of the assignment was used to group the participants into two blocks and therefore ensure better homogeneity across the two groups involved in the experiment (Section 12.2.4) for the first experiment. In the second experiment, the participants were grouped into four blocks according to the result of the first two labs of this course, one of which was about identifying inconsistency between a use case model and its corresponding UML analysis models and the other was designed for the students to practice metamodeling using the UML class diagram notations.

Two applications, namely a Video Store (VS) system and a Car Parts Dealer (CPD) system are used as experimental materials. These two systems are of similar complexity

in terms of the number of use cases in their use case models, the number of classes in their class diagrams, and the complexity of each UCS. Experimental materials must necessarily be of limited complexity since we have to consider whether the participants are able to finish the prescribed tasks, being described in Section 12.2.4, within two (Experiment 1) and four (Experiment 2) 3-hour long laboratory sessions.

The experiments were part of a series of compulsory laboratory exercises that were part of the course curriculum. After the experiments, the participants were asked to sign a consent form to indicate whether they allowed their laboratory results to be used for research purposes. The experiment plan had been reviewed and received clearance through the Carleton's Research Ethics Committee before collecting the consent forms. Participants who participated in the experiments signed a consent form to confirm their agreement on our using of the collected data for research purposes.

12.2.3 Hypotheses formulation

In this section, we only formulate experimental hypotheses for our second research goal as the first one exclusively focuses on characterizing the ease with which developers can apply the restriction rules and does not involve a comparison. The experiment for Goal 2 (Experiment 2) has one independent variable *Method*, with two treatments: *UCM_R* and *UCM_UR*, respectively denoting the use or not of RUCM, and four dependent variables *CD*, *SD*, *CS*, and *QC*, respectively denoting the quality of analysis class diagrams, the quality of analysis sequence diagrams, the consistency between analysis class and sequence diagrams, and the correctness of responses to a comprehension questionnaire.

Based on the above variables, we can formulate the following null hypothesis (H_0) to be tested for each dependent variable for Goal 2: there is no significant difference between *UCM_R* (RUCM) and *UCM_UR* (with the standard template and unrestricted) in terms of *CD*, *SD*, *CS*, and *QC*. The alternative hypotheses (H_a) is then two-tailed and stated as: *UCM_R* results in different quality or consistency of analysis models, or different

correctness of responses to the comprehension questionnaire when compared to UCM_UR. Formal hypotheses are provided in Table 17.

Table 17 Hypotheses – Experiment 2

Dependent Variable	Null Hypothesis	Alternative Hypothesis
Quality of analysis class diagram	$H_0: CD(UCM_R) = CD(UCM_UR)$	$H_a: CD(UCM_R) \neq CD(UCM_UR)$
Quality of analysis sequence diagram	$H_0: SD(UCM_R) = SD(UCM_UR)$	$H_a: SD(UCM_R) \neq SD(UCM_UR)$
Consistency between analysis class and sequence diagrams	$H_0: CS(UCM_R) = CS(UCM_UR)$	$H_a: CS(UCM_R) \neq CS(UCM_UR)$
Correct response rate of the comprehension questionnaire	$H_0: QC(UCM_R) = QC(UCM_UR)$	$H_a: QC(UCM_R) \neq QC(UCM_UR)$

12.2.4 Experiment design

In this section, we report the experiment design of the first experiment in Section 12.2.4.1, followed by the rationale for conducting a replication (the second experiment) and its experiment design (Section 12.2.4.2).

12.2.4.1 Experiment 1

The participants were asked to perform two tasks over two laboratories (3 hours each). Task 1 is about defining UCMods by applying RUCM (Chapter 3). In this task, the use case diagrams of the two systems, partially filled UCSs, and a comprehension questionnaire designed for evaluating the restrictions rules, were provided as input documents to the participants. Task 2 is about the construction of analysis models from two types of UCMods, one of which applied RUCM (UCM_R) whereas the other only applied a standard template without restrictions (UCM_UR).

As stated previously, an assignment was designed to train the participants to apply RUCM. The assignment was essentially similar to Task 1 except that a different system was used and the participants were not monitored while they did their assignment. Individual feedbacks were given to each participant and a solution of the assignment was also provided before the experiment was conducted. Based on the grades of the

assignment preceding the experiment, we defined the following three blocks: grades B to A+ (15 participants), grades B- to F (13 participants), and absent (ABS) (6 participants). As shown in Table 18, the participants were then divided into two groups: A and B. Each of the two groups was then randomly assigned participants from the three blocks in nearly identical proportions.

In Lab 1, the participants in group A were asked to complete UCSs of the VS system by applying RUCM, whereas the participants in group B did the same task on the CPD system. In Lab 2, we further divided the participants of group A into groups A1 and A2, so that the participants in A1 could derive class and sequence diagrams from the UCM_R use case model for the CPD system, while the participants in A2 did the same from the UCM_UR use case model. The same strategy was followed for group B. When assigning participants to sub-groups we followed the same blocking strategy as the one used to create groups A and B. Note that we used different systems for the two labs for each group of participants to limit learning effects that would otherwise constitute a threat to validity. For example, group A used the VS system in Lab 1 but the CPD system in Lab 2. In total, 26 data points were obtained for Task 1 and Task 2 (14 data points for treatment UCM_R and 12 for treatment UCM_UR), respectively.

Table 18 Participant groups and tasks – Experiment 1

Lab	Task	Group A		Group B		Obtained data points	
		Group A1	Group A2	Group B1	Group B2		
1	Defining UCSs	VS		CPD		26	
2	Deriving analysis models	CPD in UCM_R	CPD in UCM_UR	VS in UCM_UR	VS in UCM_R	UCM_R	UCM_UR
						14	12

12.2.4.2 Experiment replication (Experiment 2)

In Lab 2 of Experiment 1, the participants were asked to derive both class and sequence diagrams. However, most of the participants were not able to finish sequence diagrams due to time constraints and therefore we were not able to evaluate the impact of RUCM on the quality of manually created analysis sequence diagrams. Besides, no statistically significant differences were observed for analysis class diagrams in terms of their completeness and redundancy in Experiment 1 and we thought it was perhaps also due to

the time constraints. Therefore, we replicated Task 2 to 1) evaluate the impact of RUCM on the quality of analysis sequence diagrams and 2) to see whether significant differences between two treatments can be identified in terms of completeness and redundancy of generated analysis class diagrams if more time is given to participants. Task 2 was split into two tasks in Experiment 2: one was to construct an analysis class diagram (Task A) and the other was to construct analysis sequence diagrams (Task B). The participants in Experiment 2 were asked to perform Task A and Task B over two consecutive laboratories (3 hours each).

Based on the average grades of two laboratories preceding the experiment, we defined the following three blocks: grades A+ (11 participants), grades A and A- (12 participants), grades B+ – C+ (9 participants), and grades C – D- (7 participants). As shown in Table 19, the participants were then divided into four groups: G1, G2, G3, and G4. Each of the four groups was then randomly assigned participants from the four blocks in nearly identical proportions.

In Lab 1, the participants in group G1 were asked to derive analysis class diagrams from the use case model with restrictions for the CPD system, while the participants in group G2 did the same from the use case model without restrictions. The same strategy was followed for groups G3 and G4. In Lab 2, the participants were provided the same use case model as the one they were assigned in Lab 1 but they were asked to derive analysis sequence diagrams and at the same time keep consistency between these sequence diagrams and the class diagrams they designed in Lab 1. The participants were also told to refine their class diagrams designed in Lab 1 whenever necessary. At the end of the lab, the participants were asked to submit their original class diagram from Lab 1, refined class diagram, and sequence diagrams from Lab 2. In Lab 3 and Lab 4, each participant group performed Task A and Task B with a different system and a different treatment. As a result, each group executed different combinations of treatment and system and therefore learning effects that would constitute a threat to validity were limited.

Table 19 Participant groups and tasks – Experiment 2

Lab	Task	G1	G2	G3	G4
1	Deriving class diagram (Task A)	CPD in UCM_R	CPD in UCM_UR	VS in UCM_R	VS in UCM_UR
2	Deriving sequence diagram (Task B)	CPD in UCM_R	CPD in UCM_UR	VS in UCM_R	VS in UCM_UR
3	Deriving class diagram (Task A)	VS in UCM_UR	VS in UCM_R	CPD in UCM_UR	CPD in UCM_R
4	Deriving sequence diagram (Task B)	VS in UCM_UR	VS in UCM_R	CPD in UCM_UR	CPD in UCM_R

12.2.5 Instrumentation

The instruments of an experiment are classified into three types: experiment objects, guidelines and measurement instruments [108]. In this section we discuss our experiment instruments by conforming to this classification.

12.2.5.1 Task 1

Use cases are specified by applying RUCM (Chapter 3). Inputs are the use case diagrams of the two systems, the corresponding UCSs, and a comprehension questionnaire designed to capture the participants' subjective opinions on the restrictions rules. These input documents are further discussed below. The UCSs completed by the participants and their responses to the comprehension questionnaire were collected and used to evaluate the application of each restriction rule.

Experiment objects

For each system (CPD and VS), the experiment objects are composed of a use case diagram, a set of UCSs, and a system description. These two systems were originally designed as the lab materials for the Software Engineering course (4th year undergraduate course) and have been used for several years. Each document contains a textual system description, a use case diagram along with UCSs following a standard template [14], a class diagram, and sequence diagrams for a subset of the use cases. The use case diagrams and the textual system descriptions are reused as experiment objects without making any changes. Since Task 1 requires that the participants document UCSs by

applying RUCM, we provided the participants the partially filled UCSs complying with our use case template. The UCSs were partially filled and contain descriptions for only the following fields of the template (Section 3.1.1): *Use Case Name*, *Brief Description*, *Primary Actor*, *Secondary Actor*, *Dependency*, and *Generalization*. The rationale was to provide an overview of high level requirements (e.g., thanks to the brief description), ensure UCSs were consistent with the use case diagram, and let the participants focus on defining flows of events, which is the most complex part of UCSs and on which most of our restriction rules apply.

Experiment guidelines

A lab description was provided to the participants at the beginning of each lab, describing the list of documents provided, the task of the lab, and the submission guidelines. The participants were asked to complete five UCSs during the three-hour lab. The lab description also made it clear that the restriction rules had to be applied and this was a very important component of the evaluation of lab results.

Measurement instruments

An evaluation questionnaire was designed for the participants to characterize each restriction rule according to three measures: *Understandability*, *Applicability*, and *Restrictiveness*. Three questions or statements were designed to capture these three measures on an appropriate scale (Table 20). The first question is a “yes/no” question to capture whether the participants perceived they were able to understand each restriction rule and were able to properly apply them at the time when the lab task was performed. The second statement is evaluated on a four-point Likert scale [79] question, which requires the participants to rate each restriction rule according to the extent to which they perceive it to be straightforward to apply. The third statement is also defined on a four-point Likert scale. It is used to capture the perceived restrictiveness of each restriction rule. The complete questionnaire is provided in Appendix I for reference.

Table 20 Comprehension questionnaire of Task 1

Measure	Question/Statement	Scale
<i>Understandability</i>	I understood the restriction rule and was able to properly apply it. (Question asked for each rule.)	Yes / No
<i>Applicability</i>	The restriction rule is straightforward to apply. (Question asked for each rule.)	4-point Likert scale: Completely agree, Generally agree, Generally disagree, Completely disagree
<i>Restrictiveness</i>	The restriction rule was too restrictive. (Question asked for each rule.)	4-point Likert scale: Completely agree, Generally agree, Generally disagree, Completely disagree

12.2.5.2 Task 2

Task 2 consists in deriving analysis models from two types of use case models: One use case model documented in RUCM (UCM_R) (e.g., groups A1 and B1 in Experiment 1—Section 12.2.4) whereas the other only uses a standard template [14] (UCM_UR) (e.g., groups A2 and B2). Notice that the participants were equally trained to understand our use case template and the standard template. With the provided use case models, in Experiment 1, the participants were asked to manually derive an analysis model (class and sequence diagrams) and also answer a comprehension questionnaire to assess their understanding of the use cases. In Experiment 2, the participants were asked to derive a class diagram for two different systems in Lab 1 and Lab 3 (Task A), and derive corresponding sequence diagrams and answer a comprehension questionnaire to assess their understanding of the use cases in Lab 2 and Lab 4 (Task B).

The standard template [14] of UCM_UR that we used as a comparison baseline to document UCSs shares common fields with our template: use case name, brief description, precondition, primary actor, secondary actors, dependency, and generalization. However, it is missing the following characteristics (which are specified as restriction rules in RUCM):

- It does not require that each flow of events (both basic flow and alternative flow) of a UCS contains its own postcondition.
- It does not distinguish different types of alternative flows.

- It does not require that the action steps of a UCS match one of the five interaction patterns we specified in Section 4.2.3.
- It does not enforce using any keyword to clearly specify interactions between the basic flow and its corresponding alternative flows, contrary to our template.

Experiment objects

The CPD and VS systems come in two versions for Task 2: they contain the same use case diagram but have different UCSs (with or without restrictions). Both sets of UCSs were specifically created for the experiment and carefully reviewed by the authors to ensure that they both contain the same amount of information though through different forms so that the participants have identical chances of creating identical analysis models from them.

Experiment guidelines

As for Task 1 (Section 12.2.5.1), in Task 2, we also provided the participants a lab description at the beginning of the lab, describing the list of documents provided, the task of the lab, and the submission guidelines. With the use case models as input documents, the participants were asked to design a class diagram. We made it clear in the lab description (Experiment 1: Lab 2, Experiment 2: Lab 1 and Lab 3) that the participants should, based on the use case description, assign meaningful names for each class, attribute, and operation, and apply the well-established *Entity/Boundary/Control* stereotype classification [15] for each class. The participants were also asked to complete a comprehension questionnaire during the lab (Experiment 1: Lab 2, Experiment 2: Lab 2 and Lab 4), which was designed to evaluate how well they were able to understand the flow of events of each UCS. In Experiment 1, Lab 2, the participants were also asked to derive sequence diagrams for two selected use cases; however most of the participants were not able to derive (complete) diagrams due to time constraints, and we therefore decided not to analyze them. In Experiment 2, deriving sequence diagrams (Task B) was

completed in Lab 2 and Lab 4. The participants working on the CPD (VS) system were asked to derive three (two) sequence diagrams for three (two) selected use cases. Participants worked on different numbers of use cases to ensure balanced tasks given the differences of the complexity of UCSs: the three CPD use cases had the similar overall complexity as the two VS use cases. We measured the complexity of a use case by simply calculating the total number of condition and action sentences contained the use case specification.

Measurement instruments

A comprehension questionnaire was designed for each system to quickly evaluate, in a repeatable and unbiased way, the extent to which a participant understood the main body (flows of events) of each UCS. To avoid introducing any bias, we ensured comprehension questions were answerable by both the participants using the restricted and unrestricted use case models.

Questions in the comprehension questionnaires are grouped per use case so that the questionnaires can be easily browsed. The complete questionnaires for the two systems are respectively provided in Appendix J and Appendix K. Each question covers either one of the following aspects of UCSs: action sequences, object responsibilities, conditions triggering actions, and general comprehension. An example question for each aspect is provided in Table 21. Each multiple-choice question includes two choices, namely “Not specified in the UCS” and “Other answer”, so that questions are at the same time closed and open, thereby allowing us to collect complete information. (For instance, some ambiguities in UCSs may prevent the participants from selecting one of the available choices, in which case they can use either one of these two specific choices.) For each question, the participants were also asked to indicate where they found relevant information to answer the question (location). These two special choices, along with the location question, should also significantly reduce the chances that participants randomly select a choice. During data collection, we checked consistency between the selected choice and the location answer: no inconsistency was observed. For each use case, a

general comprehension question was asked to determine whether the participants identified any ambiguity in the corresponding UCS. The participants' responses to this question were carefully verified to determine the validity of the data collected, but were not directly used to test our experimental hypotheses.

All multiple-choice questions contain one and only one correct answer, except for one question in the comprehension questionnaire of the VS system, where two choices are partially correct and together make up a complete, correct answer to the question. Therefore, when grading answers to this question we considered both choices as correct and we also considered "other" as correct if the correct explanation were provided. If a response to a multiple-choice question was "Not specified in the UCS", we assumed that there either exists an ambiguity in the UCS, which further leads to a low quality analysis model, or the participants were not able to correctly answer the question. Such an answer was therefore considered to be incorrect. If the response to a multiple-choice question is "Other", then the response was carefully checked to see whether it was equivalent to the correct answer.

Table 21 Classification and examples of comprehension questions

Category	Example question
Action sequences	When the part number is unknown, Sales provides an alternative part number to the system. Then what happens? 1. The system requests DBMS to check whether this alternative part number is known or unknown. 2. The system orders the part with the alternative part number for the customer. 3. The use case terminates. 4. Not specified in the use case specification. 5. Other answers: Where did you get the information from?
Object responsibilities	Who is responsible to create a pending order if the customer does not accept the alternative order provided by the system? 1. The system 2. Customer 3. Sales 4. DBMS 5. Receiving&Shipping 6. Accounting 7. Not specified in the use case specification. 8. Other answers: Where did you get the information from?
Conditions	Under which condition is use case Video Overdue invoked? (If you cannot find the answer directly from the use case specification, please indicate it. If you find the answer in the use case specification, please indicate the place.)
General comprehension question	Did you identify any place(s) in the use case specification, which causes any confusion for you to understand the use case, for example, in terms of the sequence of the steps of the basic flow or the alternative flow(s), the subjects of actions? If your answer is "yes", please list the places and provide a brief explanation.

Questionnaire characteristics for the two systems are summarized in Table 22. In total, fifteen questions contribute to the measurement of the comprehension questionnaire for the CPD system; while twenty-five questions contribute to that of the VS system. The difference in the numbers of the questions is simply due to the different numbers of UCSs of the two systems.

Table 22 Question distributions of the comprehension questionnaires for Task 2

	CPD	VS
# of multiple-choice questions	15	23
# of non multiple-choice questions	0	2
# of general questions (not used to test our experimental hypotheses)	6	5
Total number of questions used to test our experimental hypotheses	15	25

12.2.6 Evaluation measurement and data collection

The goal of Task 1 is to evaluate each restriction rule based on four measures: *Error Rate*, *Understandability*, *Applicability*, and *Restrictiveness*. The goal of Task 2 is to evaluate the impact of either our restriction rules and template or a standard template on the quality of manually derived analysis models, with two dependent variables: the quality of class diagrams (abbreviated as CD), sequence diagrams (abbreviated as SD), the consistency of class and sequence diagrams (CS), and the correctness of responses to the comprehension questionnaires (abbreviated as QC). All these measures and their data collection are described next.

12.2.6.1 Error Rate

The *Error Rate* of a restriction rule represents the rate of improper applications of the rule, on the scale from 0 to 1. Error rates are presented in Table 24 (first 16 rules) and Table 25 (last 10 rules), and are based on raw data as measured according to metrics in Table 23. The first column of Table 24 and Table 25 lists the restriction rule number. The second column describes the formulas used to calculate the *Error Rate* of each rule. In Table 23, the first column provides the names of the metrics and the second column specifies the metrics or their calculations. The variable $N_{m,r}$ specific to the formulas (Table 25, Column 2) for rule r (one of the rules R17-R25) is defined in Table 25, Column 3.

Table 23 Measures used to derive Error Rate

Measure	Specification (a single UCS)
N_{v_r}	# of violations of a specific restriction rule r
N_{missed_r}	# of instances where the keyword (defined as the restriction rule r) should be applied, but is not
N_{ASteps}	Total number of action steps (sentences)
N_{Cond}	Total number of condition sentences: precondition, postcondition, IF/ELSEIF condition, VALIDATES THAT condition, and UNTIL condition
$N_{Include}$	Total number of INCLUDE USE CASE sentences
N_{Extend}	Total number of EXTENDED BY USE CASE sentences
$N_{AltFlows}$	Total number of alternative flows
N_{NP}	Total number of noun phrases
N_{IF}	Total number of IF-THEN-ELSE-ELSEIF-ENDIF conditional logic sentences
N_{ABORT}	Total number of ABORT sentences
N_{RESUME}	Total number of RESUME STEP # sentences
$N_{MEANWHILE}$	Total number of MEANWHILE sentences
N_{VALTS}	Total number of VALIDATES THAT sentences
$N_{DO-UNTIL}$	Total number of DO-UNTIL sentences
N_{RFS}	Total number of the places where the keyword RFS is applied
N_{Spe}	$=N_{ABORT}+N_{RESUME}$
N_{Dep}	$=N_{Include}+N_{Extend}$
$N_{NormalA}$	$=N_{ASteps}-N_{Spe}-N_{Dep}$
$N_{NormalS}$	$=N_{NormalA}+N_{Cond}$
N_{AHS}	$=N_{ASteps}+N_{Cond}-N_{VALTS}$

The *Error Rate* of each restriction rule equals (Table 24) the number of violations of the rule r (N_{v_r}) divided by the total number of steps where the rule could be applied, i.e., either $N_{NormalA}$, N_{ASteps} , N_{NP} , or $N_{NormalS}$. For example, R1 is specified as “The subjects of sentences in basic and alternative flows should be the system or actors”. It puts restriction on action steps, except those describing use case relationships (i.e., those that use keywords INCLUDE and EXTEND) and those specifying what happens at the end of an alternative flow (i.e., those that use keywords ABORT and RESUME). Therefore the error rate of R1 for a UCS equals $N_{v_1}/N_{NormalA}$, where N_{v_1} is the number of violations of R1 in the UCS and $N_{NormalA}$ (Table 23) is the total number of action steps (N_{ASteps}) of the UCS that do not contain keywords ABORT, RESUME, INCLUDE USE CASE and EXTENDED BY USE CASE (we subtract N_{Spe} and N_{Dep} from N_{ASteps} in Table 23). Some restriction rules require two error rate measures. For example (Table 25), the error rate of R17 is calculated as the average of N_{missed_17}/N_{Alls} and $N_{m_17}/N_{Include}$, where N_{missed_17}/N_{Alls} captures the rate of absence of keyword INCLUDE USE CASE (specified in R17), i.e., it should be used but it is not, whereas $N_{m_17}/N_{Include}$ captures the rate of erroneous

occurrences (the keyword is used, but either in a wrong situation, without the included use case name, or with an incorrect use case name, or should not be applied at all).

Table 24 Error Rate measurements (R1–R16)

#	Measure	#	Measure	#	Measure	#	Measure
R1	$N_{v_1}/N_{NormalA}$	R5	$N_{v_5}/N_{NormalA}$	R9	N_{v_9}/N_{NP}	R13	$N_{v_13}/N_{NormalS}$
R2	N_{v_2}/N_{ASteps}	R6	$N_{v_6}/N_{NormalA}$	R10	$N_{v_10}/N_{NormalS}$	R14	N_{v_14}/N_{NP}
R3	$N_{v_3}/N_{NormalA}$	R7	$N_{v_7}/N_{NormalA}$	R11	$N_{v_11}/N_{NormalS}$	R15	$N_{v_15}/N_{NormalS}$
R4	N_{v_4}/N_{ASteps}	R8	$N_{v_8}/N_{NormalS}$	R12	$N_{v_12}/N_{NormalS}$	R16	N_{v_16}/N_{NP}

Table 25 Error Rate measurements (R17–R26)

#	Measure		Description (N_{m_r})—see Table 9 for the corresponding rules/keywords
R17	$(N_{missed_17}/N_{AllS} + N_{m_17}/N_{Include})/2$	+	# of instances where the keyword is applied, but either in a wrong situation, without the included use case name, or with incorrect use case name, or should not be applied at all.
R18	$(N_{missed_18}/N_{AllS} + N_{m_18}/N_{Extend})/2$	+	# of instances where the keyword is applied, but either in a wrong situation, without the extended use case name, or with incorrect use case name.
R19	$(N_{missed_19}/N_{AltFlows} + N_{m_19}/N_{RFS})/2$	+	# of instances where the keyword is applied, but either in a wrong situation, without the basic flow step number followed, or with a wrong basic flow step number.
R20	N_{m_20}/N_{IF}		# of instances where the keyword is incorrectly applied, such as incorrect grammar.
R21	$(N_{missed_21}/N_{AllS} + N_{m_21}/N_{MEANWHILE})/2$	+	# of instances where the keyword is applied, but in a wrong situation, or the action steps connected by the keyword are not appropriate (e.g., non-action steps).
R22	N_{m_22}/N_{VALTS}		# of instances where the alternative case is not described in its corresponding alternative flow, the condition sentence is not a complete sentence, or there is no condition sentence followed by the keyword.
R23	$(N_{missed_23}/N_{AllS} + N_{m_23}/N_{DO-UNTIL})/2$	+	# of instances where the keyword is applied, but in a wrong situation, or the grammar is not followed: e.g., missing condition.
R24	$(N_{missed_24}/N_{AllS} + N_{m_24}/N_{ABORT})/2$	+	# of instances where the keyword is applied in a wrong place: e.g., not the last step of an alternative flow.
R25	$(N_{missed_25}/N_{AllS} + N_{m_25}/N_{RESUME})/2$	+	# of instances where the keyword is applied, but not in an alternative flow, or the keyword is applied in an alternative flow, but not in the last step of an alternative flow.
R26	$N_{v_26}/(N_{AltFlows}+1)$		$N_{AltFlows}+I$: # of the flows of events of a UCS.

For each UCS, the error rate of a specific restriction rule r ($ErrorRate_r$) is:

$$ErrorRate_r = \frac{\sum_{s=1}^{|S|} (\sum_{u_s=1}^{|U_s|} ErrorRate_{u_s})}{U \times |S|}$$

where S is the set of participants (s is an index identifying each participant), U_s is the set of UCSs created by participant s (u_s is an index identifying each UCSs created by participant s), U denotes the total number of UCSs written by all the participant.

12.2.6.2 Understandability, Applicability and Restrictiveness

Understandability is one of the three subjective measures used to assess the restriction rules and is based on responses to the first (yes/no) question of the comprehension questionnaire of Task 1 (Section 12.2.5.1). This measure, normalized on the scale from 0 to 1, is the ratio of “yes” in all collected responses. The other two subjective measures, *Applicability* and *Restrictiveness*, are measured by using the participant responses to the second and third questions of the comprehension questionnaire of Task 1 (Section 12.2.5.1), respectively. Recall that both questions are answered on four-point Likert scales, from 1 (Completely disagree) to 4 (Completely agree). Our analysis will focus on comparing average scores across all participant responses.

12.2.6.3 Quality of analysis class diagram (CD)

The quality of an analysis class diagram is evaluated from three aspects: *Correctness*, *Completeness*, and *Redundancy*. The reference class diagrams, used as the basis to evaluate class diagrams designed by the participants, are the original class diagrams of the two case study systems we used in Task 2 (Section 12.2.5.2). Data are collected for the first five measures in Table 26 from the reference class diagrams (e.g., number of classes in each reference class diagram (N_{class})); while data are also collected for the last 10 measures in Table 26 from the class diagram of each participant. All these data are then used to compute the measures of *Completeness*, *Correctness* and *Redundancy* of a participant class diagram according to the formulas presented in Table 28. The completeness of a class diagram (R_{CDcomplt}) is computed as the average of Class Completeness (R_{clp1}), Association Completeness (R_{clp2}) and Generalization Completeness (R_{clp3}) since we consider classes, associations and generalizations to be the three most important (structural) aspects of a class diagram. The class diagram correctness ($R_{\text{CDcorrect}}$)

is determined by the correctness of classes (R_{Ci})—computed as the average, over the complete class diagram, of the class measures of *Completeness* (R_{Ccompl}) and *Correctness* ($R_{Ccorrect}$) (Table 27)—and associations. The class diagram redundancy (R_{CDre}) is computed as the ratio of redundant classes (N_{re}) over all the classes of a participant’s class diagram (N_{class}).

Table 26 Measures used to derive CD

#	Measure	Specification
1	N_{class}	# of classes in the reference model
2	N_{asso}	# of associations in the reference model
3	N_{gen}	# of generalizations in the reference model
4	N_{attr}	# of attributes in entity classes in the reference model
5	N_{opr}	# of operations of control or boundary classes in the reference model
6	N_{cp1}	# of missing attributes of a class
7	N_{cp2}	# of missing operations of a class
8	N_{clp1}	# of missing classes in a class diagram
9	N_{clp2}	# of missing associations in a class diagram
10	N_{clp3}	# of missing generalizations in a class diagram
11	N_c	# of matching classes in a class diagram
12	N_{class}	# of classes in a class diagram
13	N_{clr2}	# of incorrect associations of a class diagram
14	N_a	# of matching associations between matching class diagrams
15	N_{re}	# of extra classes that are redundant, excluding equivalent model elements

For each class of the reference class diagram of a case study system, we look for a class with the same name in a participant class diagram. If such a matching class is found, then it is evaluated according to the quality measures for a class (Table 27); otherwise, we keep looking for a design equivalent¹⁵ to the reference class in the participant class diagram. If no such equivalent design exists, then we consider the reference class as missing and therefore the participant diagram as incomplete. When all the reference classes have been looked at, we obtain three outputs: 1) A set of matching classes which are evaluated by using the quality measures for a class (Table 27); 2) A set of equivalent designs, which are not measured because this would require either a subjective measurement or a large number of specific measures. Besides not many such equivalent designs have been found and not measuring them does not really impact the measurement of CD; 3) A set of reference classes, missing in the participant class diagram. A

¹⁵ An equivalent design may contain one or more model elements, which could be attributes, multiple classes connected by associations, etc.

procedure similar to this identification of matching classes, missing classes, and equivalent class designs is also applied to identify matching/missing/equivalent attributes, operations, associations, and generalizations. Each participant class diagram is evaluated by applying the quality measures for a class diagram (Table 28).

Table 27 Quality measures for a class

Category	Measure	Formula
Completeness (R_{Ccompl})	Missing stereotype (R _{cp1})	Entity classes: $R_{Ccompl} = 1 - (R_{cp1} + R_{cp2})/2$ Boundary classes: $R_{Ccompl} = 1 - (R_{cp1} + R_{cp3})/2$
	Missing attributes $R_{cp2} = N_{cp1} / N_{atr}$	
	Missing operations $R_{cp3} = N_{cp2} / N_{opr}$	
Correctness (R_{Ccorrect})	Incorrectly named (R _{cr1})	Entity classes: $R_{Ccorrect} = 1 - (R_{cr1} + R_{cr2} + R_{cr3} + R_{cr4} + R_{cr5} + R_{cr6})/6$ Boundary classes: $R_{Ccorrect} = 1 - (R_{cr1} + R_{cr2} + R_{cr3} + R_{cr5} + R_{cr6})/5$ If the entity class under evaluation does not contain any attributes, then: $R_{Ccorrect} = 1 - (R_{cr1} + R_{cr2} + R_{cr3} + R_{cr5})/4$ If the boundary class under evaluation does not contain any operations, then: $R_{Ccorrect} = 1 - (R_{cr1} + R_{cr2} + R_{cr3} + R_{cr6})/4$
	Incorrectly stereotyped (R _{cr2})	
	Incorrectly assigned "abstract" (R _{cr3})	
	Does not represent one and only one logical concept (R _{cr4})	
	Not given a cohesive set of responsibilities (R _{cr5})	
	Does not represent the intended meaning of the class (R _{cr6})	
Class i (R_{ci})	$R_{ci} = (R_{Ccompl} + R_{Ccorrect})/2$	

Table 28 Quality measures for a class diagram

Category	Measure	Formula
Completeness (R_{CDcompl})	Class Completeness $R_{clp1} = 1 - N_{clp1} / N_{class}$	$R_{CDcompl} = (R_{clp1} + R_{clp2} + R_{clp3})/3$
	Association Completeness $R_{clp2} = 1 - N_{clp2} / N_{asso}$	
	Generalization Completeness $R_{clp3} = 1 - N_{clp3} / N_{gen}$	
Correctness (R_{CDcorrect})	Class Correctness $R_{clr1} = 1 - \sum_{i=1}^{N_c} R_{ci} / N_c$	$R_{CDcorrect} = (R_{clr1} + R_{clr2})/2$
	Association Correctness $R_{clr2} = N_{clr2} / N_a$	
Redundancy (R_{CDre})	$R_{CDre} = N_{re} / N_{eclass}$	

12.2.6.4 Quality of analysis sequence diagram (SD) and consistency of analysis class and sequence diagrams (CS)

Five measures evaluate the quality of a sequence diagram (SD): *SD Completeness*, *SD Correctness*, *SD Redundancy*, *BCE Consistency* (consistency with the Boundary/Control/Entity principle), and *SDCD Consistency*. They are defined (Table 30) based on simpler measures (Table 29). To evaluate the quality of sequence diagrams produced by participants we need reference sequence diagrams. Unfortunately we do not find sequence diagrams for the CPD and VS systems in reference textbooks or other resources. We only possess sequence diagrams manually created by the authors of this article. However, we can use sequence diagrams generated by the CASE tool aToucan (Chapter 8), which automatically generates analysis class and sequence diagrams from RUCM requirements, since these diagrams have been shown to be of high quality, the quality of renowned textbooks. Data are collected from the reference sequence diagrams to compute the first four measures in Table 29; while data are collected from the sequence diagrams being evaluated for the last 14 measures.

As opposed to the first four aspects, which compare a participant's sequence diagram to a reference sequence diagram, *SDCD Consistency* (R_{conSDCD}) measures the consistency between a participant sequence diagram and this participant's class diagram. Any violation of the following three rules contributes to $N_{\text{inconSDCD-t}}$, which determines *SDCD Consistency* (Table 30):

- Each object in a sequence diagram must be an instance of a class in the class diagram;
- Each operation, attribute in a sequence diagram must appear in the class diagram and must be consistent with it (e.g., an operation in a message must be declared in the class that is the type of the target lifeline);
- Two objects can communicate with messages provided there is an association between their respective classes.

As shown in Table 30, the *completeness* of a sequence diagram ($R_{SDcompl}$) is computed as the average of the *Message Completeness* (R_{conMsg}), *IU (InteractionUse) Completeness* (R_{conIU}), and *CF (CombinedFragment) Completeness* (R_{conCF}). The consistency of a sequence diagram ($R_{SDcorrect}$) is determined by the consistency of messages (R_{corMsg}), interaction uses (R_{corIU}), combined fragments (R_{corCF}), and message ordering (R_{corSeq}) since these are considered to be important aspects in a sequence diagram. The redundancy of a sequence diagram (R_{SDre}) is computed as the ratio of redundant messages ($N_{reMsg-t}$) over all the messages of a student's sequence diagram (N_{msg-t}). Measure *BCE Consistency* (R_{conBCE}) evaluates whether a student's sequence diagram conforms to the Boundary-Control-Entity (BCE) design principle of analysis sequence diagrams [15]. Any violation of the following three rules is considered to be a violation of the BCE principle and therefore contributes to measure N_{vBCE-t} (Table 29):

- A message is sent from a Boundary object to an Entity object.
- A message is sent from an Entity object to a Control object.
- A message is sent from an Entity object to a Boundary object.

Table 29 Measures used to derive SD

#	Measure	Specification
1	N_{msg-r}	# of messages in the reference sequence diagram
2	$N_{lifeline-r}$	# of lifelines in the reference sequence diagram
3	N_{IU-r}	# of interaction uses in the reference sequence diagram
4	N_{CF-r}	# of combined fragments in the reference sequence diagram
5	N_{msg-t}	# of messages in a tested sequence diagram
6	$N_{lifeline-t}$	# of lifelines in a tested sequence diagram
7	N_{IU-t}	# of interaction uses in a tested sequence diagram
8	N_{CF-t}	# of combined fragments in a tested (participant) sequence diagram
9	N_{mMsg-t}	# of missing messages in a tested sequence diagram
10	N_{mIU-t}	# of missing interaction uses in a tested sequence diagram
11	N_{mCF-t}	# of missing combined fragments in a tested sequence diagram
12	N_{iMsg-t}	# of occurrences of incorrect messages (i.e., incorrect message name, incorrect lifelines, wrong direction, and/or incorrect message types) in a tested sequence diagram
13	N_{iIU-t}	# of occurrences of incorrectly-applied interaction uses in a tested sequence diagram
14	N_{iCF-t}	# of occurrences of incorrectly-applied combined fragments 1. InteractionOperand is not provided or a wrong operand is used. 2. Condition is not provided or a wrong condition is used. 3. Lifelines are not correctly covered. 4. Wrong messages are covered.
15	N_{iSeq-t}	# of occurrences of incorrect sequences of messages, interaction uses, or combined fragments.
16	N_{vBCE-t}	# of violations of the Boundary-Control-Entity (BCE) principle
17	$N_{reMsg-t}$	# of extra messages that are redundant; the semantics of these messages have been modeled by other elements
18	$N_{inconSDCD-t}$	# of occurrences of inconsistency between a tested sequence diagram and its corresponding class diagram

Table 30 Quality measures for a sequence diagram

Category	Measure	Formula
Completeness ($R_{SDcomplt}$)	Message Completeness $R_{conMsg} = 1 - \frac{N_{mMsg-t}}{N_{msg-r} - N_{uMsg-r} - N_{aaMsg-r}}$	$R_{SDcomplt} = \frac{R_{conMsg} + R_{conIU} + R_{conCF}}{3}$
	IU Completeness $R_{conIU} = 1 - N_{mIU-t} / (N_{IU-r})$	
	CF Completeness $R_{conCF} = 1 - N_{mCF-t} / (N_{CF-r})$	
Correctness ($R_{SDcorrect}$)	Message Correctness $R_{corMsg} = 1 - \frac{N_{iMsg-t} + N_{aaMsg-t}}{N_{msg-t}}$	$R_{SDcorrect} = \frac{R_{corMsg} + R_{corIU} + R_{corCF} + R_{corSeq}}{4}$
	IU Correctness $R_{corIU} = 1 - N_{iIU-t} / (N_{IU-t})$	
	CF Correctness $R_{corCF} = 1 - N_{iCF-t} / (N_{CF-t})$	
	Message Sequence Correctness $R_{corSeq} = 1 - N_{iSeq-t} / N_{msg-t}$	
Redundancy (R_{SDre})	$R_{SDre} = N_{reMsg-t} / N_{msg-t}$	
BCE Consistency (R_{conBCE})	$R_{conBCE} = 1 - N_{vBCE-t} / N_{mMsg-t}$	
SDCD Consistency ($R_{conSDCD}$)	$R_{conSDCD} = 1 - N_{inconSDCD-t} / (N_{msg-t} + N_{lifeline-t})$	

12.2.6.5 Correctness of responses to the comprehension questionnaires (QC)

As discussed in Section 12.2.5.2, data about the correctness of responses to the questions of the comprehension questionnaires of Task 2 (QC), are used to evaluate the understandability of UCSs, which is normalized between 0 and 1:

For the CPD system: $QC_{CPD} = \text{number of correct responses} / 15$

For the VS system: $QC_{VS} = \text{number of correct responses} / 25$

where the denominators are the total numbers of questions in each system questionnaire.

12.3 Experiment Results and analysis

Recall that our first goal (Section 12.2.1) is to assess each restriction rule with respect to four measures: *Error Rate*, *Understandability*, *Applicability*, and *Restrictiveness*, whereas the second goal (Section 12.2.1) is to evaluate the impact of the restriction rules and use

case template on the quality of manually derived analysis models, with four dependent variables: *CD*, *SD*, *CS*, and *QC*, respectively denoting the quality and consistency of analysis class and sequence diagrams manually generated by participants and the correctness of participants responses to a comprehension questionnaire.

In Experiment 1, among the collected 26 data points for Task 2 (Table 18), one participant's result was not included in the analysis because it was very incomplete (no class diagram was derived), thus suggesting he had not complied with instructions. One participant missed the lab for Task 2 and performed the task at home in an uncontrolled manner. We excluded this data point as well. We also excluded a data point from a participant who spent significantly more than the planned three hours (3 hours 40 mins) to finish Task 2. As a result, a total of 23 data points were used for analyzing Task 2 (14 data points for treatment UCM_R and 9 for treatment UCM_UR). Though the two systems used for the experiment may lead to different results, the number of observations does not allow us to perform a separate analysis for each of them. We, however, counter-balance their possible effect by ensuring a similar proportion of observations coming from each system, for each of the tasks.

The collected and used data points in Experiment 2 are provided in Table 31. As it is often the case in controlled experiments, unforeseen, difficult to control events lead to the exclusion of invalid data. Some observations in each lab and group were left out of the analysis because of one of the following reasons. 1) Three participants (one in G2 and two in G4) did not want to sign the consent form and therefore their data points were excluded; 2) One participant's result (Lab 2-G1) was very incomplete (no meaningful messages were derived) suggesting he did not follow instructions; 3) Seven participants' results (one in Lab 2-G1, two in Lab 2-G2, one in Lab 2-G3, one in Lab 2-G4, one in Lab 4-G1, and one in Lab 4-G2) contain only one or two sequence diagrams (two or three sequence diagrams are required); 4) Four participants (two in Lab 1-G1, one in Lab 1-G4, and two in Lab 3-G4) spent significantly less than the planned three hours (less than 2 hours) to finish Task A; 5) Two participants in G1 and G4 missed Lab 3 and Lab 2, respectively, and performed the task at home in an uncontrolled manner; 6) One

participant (Lab 3-G3) derived a class diagram which is extremely similar to the reference class diagram (e.g., identical class, attribute, and operation names); 7) One participant (G3) derived a single sequence diagram for three use cases of the CPD system in Lab 4.

Table 31 Collected and analyzed data points – Experiment 2

Lab	Collected data points				Analyzed data points			
	G1	G2	G3	G4	G1	G2	G3	G4
1	10	10	9	10	8	9	9	7
2	10	10	9	8	8	7	8	6
3	10	9	9	10	9	9	8	6
4	9	9	10	8	8	8	9	6

12.3.1 Usage of restriction rules

Each restriction rule is evaluated by four measures: *Error Rate*, *Understandability*, *Applicability*, and *Restrictiveness* (Section 12.2.6). The last three measure participants' subjective opinions on rules; while *Error Rate* objectively evaluates errors made by participants when applying each rule to UCSs during Experiment 1, Task 1. In this section, experiment results are first discussed individually in terms of each measure. Then, we synthesize these observations and try to identify correlations between the measures. This analysis is motivated by the expectation that a rule that is easy to understand and apply should be less restrictive and have a lower error rate. Last, we provide suggestions regarding future training for applying restriction rules.

12.3.1.1 Error rate

Twelve of the 26 restriction rules have an error rate above or equal to 5%: Figure 37. Six other rules have nearly no errors. All 26 error rates are provided in Appendix L, Table 62 for reference. Notice that the highest error rate is 25% (R22), and that all but two rules (R20 and R22) have an error rate under 15%. If we now look more closely at R20 and R22, we notice that R20 and R22 are more complex than the others (Section 3.3). R20 contains three different cases, which specify how keywords are used to describe conditional logic sentences (IF-THEN-ELSE-ELSEIF-ENDIF) within a flow or across a reference flow and an alternative flow (e.g., IF-THEN appears in the reference flow while

ELSE-ENDIF appears in the alternative flow). We observed that the most frequently occurring errors regarding R20 involve not applying or incorrectly applying the required keywords (e.g., missing ELSE in alternative flow). R22 is also a composite and complex restriction rule, which not only specifies the usage of keyword VALIDATES THAT, but also indicates that the alternative case of the condition checking sentence containing this keyword must be described in an alternative flow.

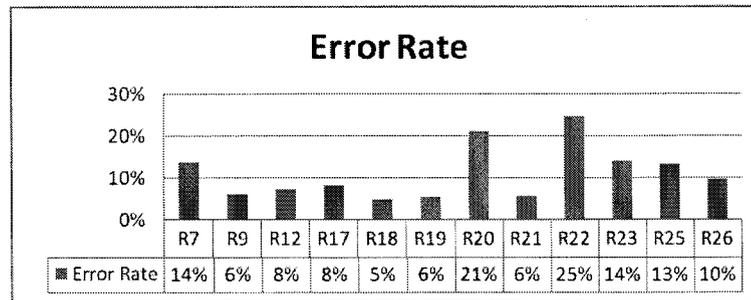


Figure 37 Error rates of restriction rules (when $\geq 5\%$)

From a more general standpoint, we can also observe from Figure 37 that most of the restrictions on the use of natural language (R1-R16) have error rates lower than the restrictions on the use of control structures specified as keywords (R17-R25). Appropriate tool support (part of our future work) can very likely be used to enforce the proper usage of keywords specified in the latter set of rules, thus potentially reducing their error rates. We also believe that more extensive training, perhaps specifically focused on error-prone rules, could further reduce error rates.

12.3.1.2 Understandability

Recall that *Understandability* of a restriction rule reflects the ease of understanding the rule according to the subjective opinions of the participants. Figure 38 presents the 11 (out of 26) rules that score below 90%. All 26 *Understandability* scores are provided in Appendix L, Table 63 for reference. The lowest score is 65% (R10 and R15). It is worth mentioning that R8, R10, R13, and R15 rely on concepts from the natural language domain (e.g., “declarative sentence”, “modal verb”, “negatively adverb”, and “participle

phrase”), which could probably explain why these four restriction rules have relatively low *Understandability* (from 65% to 75%). It is our experience that computer engineering participants in Canada have in general a limited understanding of grammatical and natural language concepts, an issue that may also extend to many IT professionals. This suggests that in the future we probably need to put more efforts on these rules during training. Another possible explanation could be that the motivation and rationale of these restriction rules is not clear to the participants since some of them (e.g., R12-R15) are designed to facilitate automated natural language processing, an aspect of the study that the participants were not informed about.

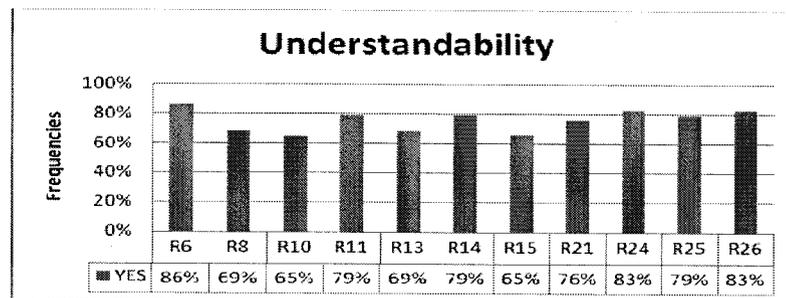


Figure 38 Understandability of the restriction rules (when <90%)

In general, we can see that the participants have received sufficient training before the experiment so that they were able to understand most of the restriction rules to a sufficient degree. Therefore, we are confident that the participants who were given use case models with restrictions in Task 2 are capable to understand the use case models and derive analysis models from them.

12.3.1.3 Applicability

Recall that *Applicability* is measured on a four-point Likert scale, from 1 (Completely disagree) to 4 (Completely agree). The frequencies of the participants’ responses on this scale are analyzed. Figure 39 presents the percentage of responses with “agree” scores on *Applicability*, showing only rules with a percentage below 90% (12 out of 26 restriction

rules). The scores for the 26 restriction rules are provided in Appendix L, Table 64, for reference.

Figure 39 shows that most of the participants (80% and above) agree that most of the restriction rules (21 rules) are easy to apply, except for rules R8, R10, R11, R13 and R15, which receive less than 80% “agree” responses. Notice that these rules also receive relatively low *Understandability* scores (Figure 38). This probably suggests that these rules are relatively difficult to understand and, as a result, they are also relatively hard to apply. As we have discussed in Section 12.3.1.2, better training and/or tool support could help improve these scores.

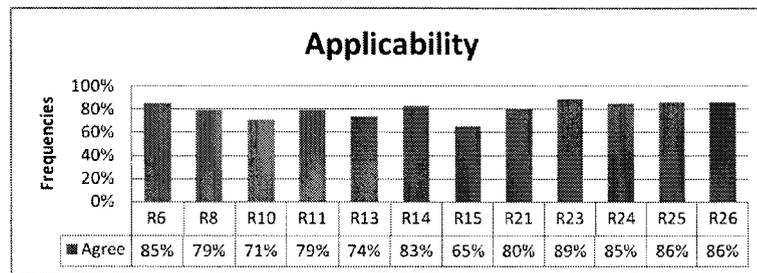


Figure 39 High applicability of the restriction rules (when < 90%)

12.3.1.4 Restrictiveness

Restrictiveness is also measured on a four-point Likert scale from 1 (Completely disagree) to 4 (Completely agree). Figure 40 presents the percentage of responses with “agree” scores on Restrictiveness, showing only the 10 (out of 26) rules receiving scores above 20%. The scores for all the 26 restriction rules are provided in Appendix L, Table 65, for reference. As shown in Figure 40, most of the participants (80% and above) agree that more than half of the restriction rules (16) are not restrictive. Though the 10 rules shown in Figure 40 all received a score above 20%, all scores are below 40% and the remaining 16 rules were below this threshold. This is not bad considering that the participants applied the restriction rules for the first time and were not informed of the rationale of the restriction rules when the experiment was conducted. In other words, if the participants knew what the rules were for, they would consider them less restrictive.

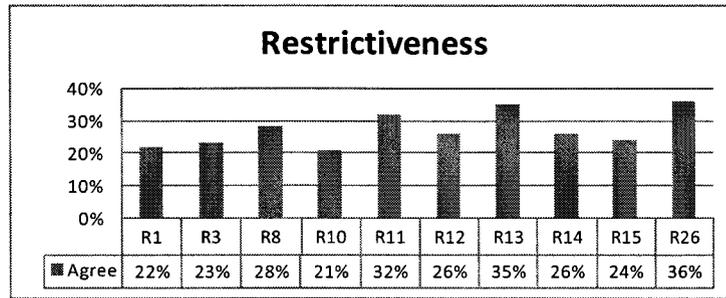


Figure 40 High restrictiveness of the restriction rules (when > 20%)

12.3.1.5 Correlation Analysis

Let us analyze the correlations between the measures of *Understandability*, *Restrictiveness*, *Applicability*, and *Error Rate*. We expect that a rule that is easy to understand and apply should be less restrictive and have a lower error rate. Spearman nonparametric correlations between each pair of the four measures are presented in Table 32. The measures of *Understandability*, *Applicability*, and *Restrictiveness* are strongly related to each other (significant correlations are identified), which means that most rules with high *Understandability* consistently have high *Applicability* and low *Restrictiveness*. However, no significant correlations between *Error Rate* and the other three measures can be identified. One possible explanation is that the participants were consistent when answering questionnaires but some of them did not gain enough insight to precisely answer those questionnaires according to their own experience of using the restriction rules: some of the participants that perceived some rules to be understandable, easy to apply and not restrictive committed mistakes when applying them while others did not.

Table 32 Spearman nonparametric correlation analysis – correlation table

Measure	By Measure	Spearman ρ	Prob> ρ
Low Restrictiveness	Understandability	0.5420	0.0042
High Applicability	Understandability	0.7894	<0.0001
High Applicability	Low Restrictiveness	0.5972	0.0013
Error Rate	Understandability	0.2930	0.1463
Error Rate	Low Restrictiveness	0.2262	0.2665
Error Rate	High Applicability	0.1299	0.5272

12.3.2 Quality of analysis models

As we have discussed in Section 12.2.3, Goal 2 involves one independent variable (*Method*) with two treatments, *UCM_R* and *UCM_UR*, respectively denoting the use or not of RUCM, and four dependent variables *CD*, *SD*, *CS*, and *QC*, respectively denoting the quality of analysis class diagrams, the quality of analysis sequence diagrams, the consistency between analysis class and sequence diagrams, and the correctness of responses to a comprehension questionnaire (Sections 12.2.6.3 and 12.2.6.4). In this section, we discuss the impact of the independent variable *Method* on the dependent variable *CD* (Section 12.3.2.1), *SD* (Section 12.3.2.2), *CS* (Section 12.3.2.3), and *QC* (Section 12.3.2.4) and also investigate the possible interactions between *Method* and two factors (*System* and *Order*) on these four dependent variables.

12.3.2.1 Quality of analysis class diagram (CD)

This section discusses the impact of *Method* (two treatments: *UCM_R* vs. *UCM_UR*) on *CD* (the quality of analysis class diagrams), which is determined by several measures (Section 12.2.6.3). Next, we investigate possible interactions between *Method* and a number of factors on *CD*: *System* (*CPD* vs. *VS*) and *Order* (2.1¹⁶ vs. 2.3). The descriptive statistics of all the experiments regarding *CD* are provided in Table 66, Appendix L for reference.

¹⁶ In the rest of the thesis, we use 2.1, 2.2, 2.3, and 2.4 to denote the first, second, third and fourth labs of Experiment 2, respectively.

One first observation is that all means for *Association Completeness*, for all experiments, are below 0.25. This indicates that less than 25% of the associations in the reference class diagrams were derived by the participants. (Recall that the reference class diagrams of CPD and VS contain 15 and 10 associations, respectively.) Furthermore, many participants were not even able to derive any of the reference associations. In such situations, *Association Correctness*, which is defined as the ratio of correctly identified associations to all identified associations and also contributes to the calculation of *CD Correctness*, is then undefined due to a division by zero. Therefore, we exclude *Association Correctness* and *CD Correctness* from our analysis.

In the rest of the section, we first report univariate analysis results for *CD* on the following four measures: *Class Completeness*, *CD Completeness*, *Class Correctness*, and *Redundancy*. Then interaction effects between *Method* and *System* and between *Method* and *Order* are discussed. Last, we summarize the analysis results.

Univariate analysis

Figure 41 to Figure 44 show, for the four quality measures of *CD*, the mean diamond plots of Experiment 1 and Experiment 2. This visual representation helps compare the means of the different measures for treatments UCM_R and UCM_UR. A mean diamond depicts the sample mean and 95% confidence interval. The line across each diamond pair represents the group mean and the vertical span of each diamond represents the 95% confidence interval. The width of each diamond represents the size of the sample. Notice that though the systems used for the experiment might lead to different results, the number of observations does not allow us to perform a separate analysis for each of them in Experiment 1. Additionally, for the purpose of comparison, the mean diamonds of the four measures in Experiment 2 (without differentiating systems and labs) are also provided in these figures.

As shown in these figures, the participants applying UCM_R consistently performed better than the participants applying UCM_UR in terms of all the four measures in both

Experiment 1 and Experiment 2. The results of a two-tailed t -test to assess the statistical significance, for each system, of the difference between the two treatments are discussed below.

By comparing the global means for the two experiments, we can notice that the results of Experiment 2 are overall better than those of Experiment 1 regarding *Class Completeness*, *CD Completeness*, and *Class Correctness*. However it is the opposite for *Redundancy*; Experiment 2 yielded higher mean values than Experiment 1. As discussed in Section 12.2.4, participants in Experiment 2 were given twice as much time to derive class diagrams and we were expecting to obtain more complete class diagrams and, as a result, more redundant classes.

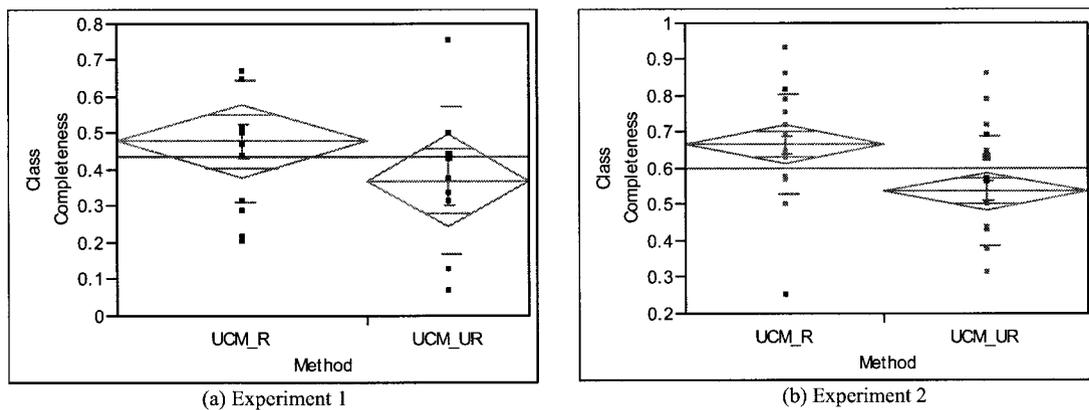


Figure 41 Distributions of Class Completeness - Labs 1&2

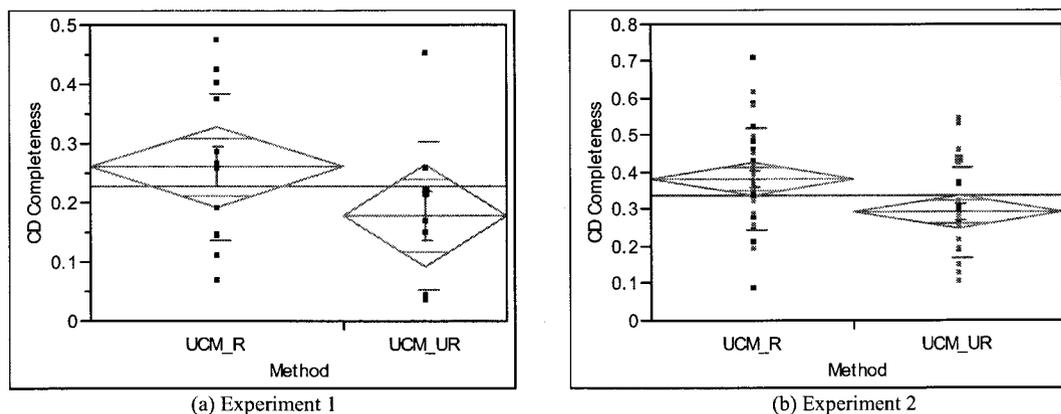


Figure 42 Distributions of CD Completeness - Labs 1&2

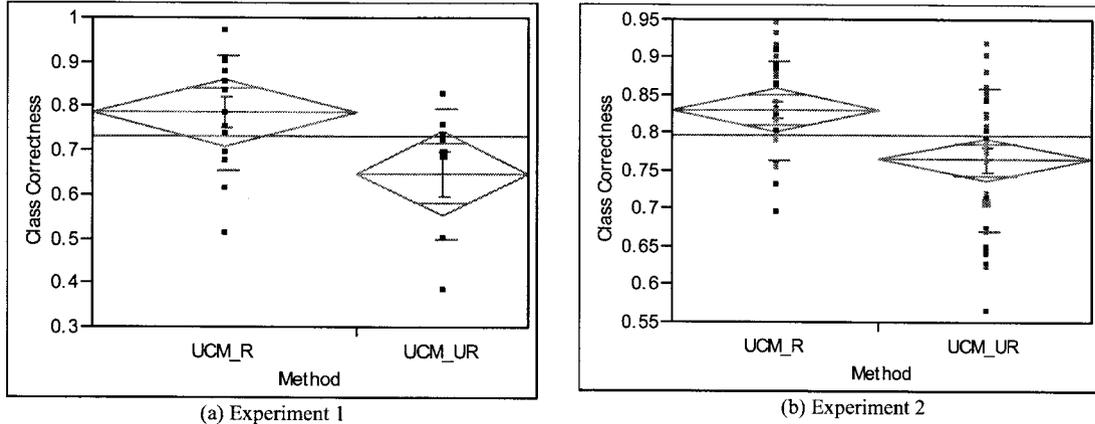


Figure 43 Distributions of Class Correctness - Labs 1&2

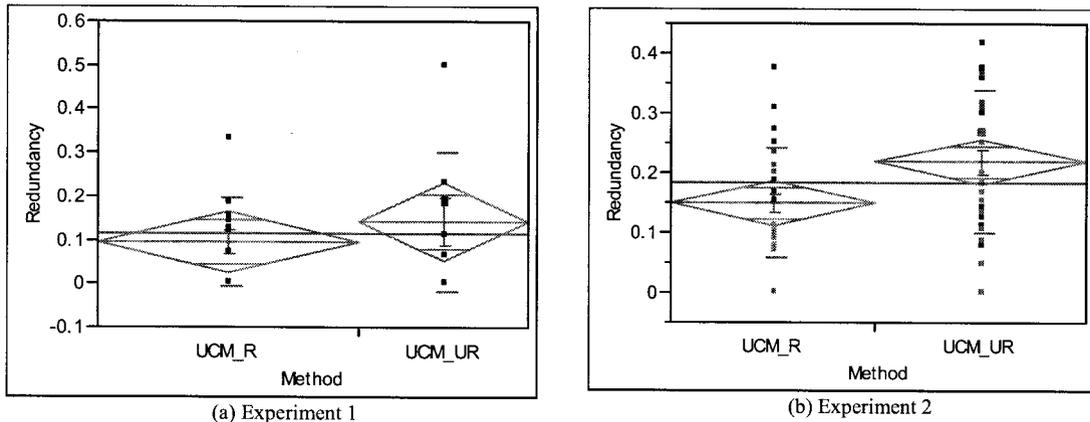


Figure 44 Distributions of Redundancy - Labs 1&2

Figure 45 to Figure 48 show mean diamonds of the four measures of lab 2.1. Figure 49 to Figure 52 show mean diamonds of the four measures of lab 2.3. In each of these figures, the mean diamonds for each system are provided for the purpose of comparison.

We notice from these figures that the participants applying UCM_R performed better than those applying UCM_UR, for both systems in both labs 2.1 and 2.3. Also, the mean values of the VS system are consistently higher (*Class Completeness*, *CD Completeness* and *Class Correctness*) or lower (*Redundancy*) than those of the CPD system. Therefore

we conducted a two-way ANOVA analysis to assess the impact of *System* (CPD vs. VS) on *CD*.

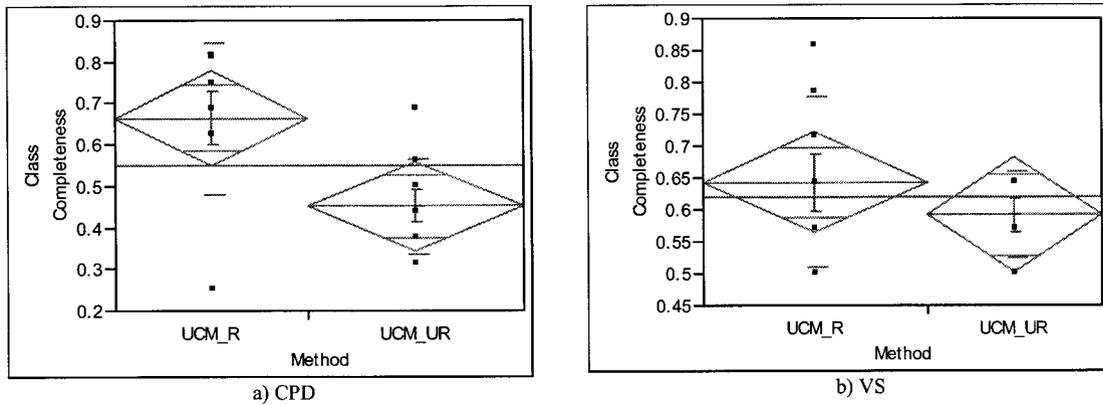


Figure 45 Distributions of Class Completeness - Lab 2.1

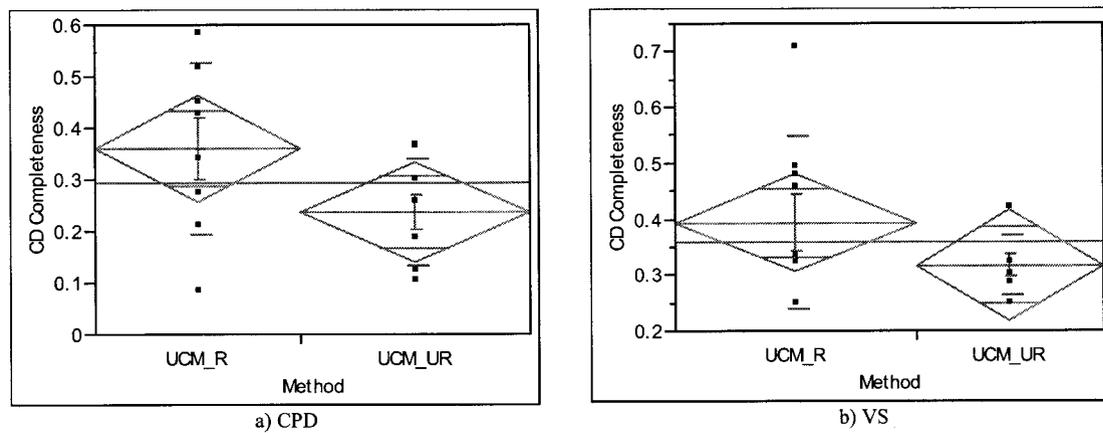
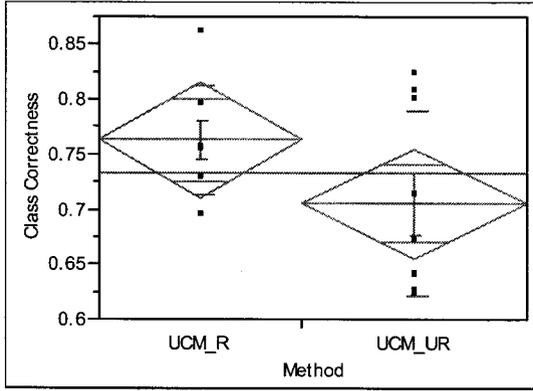
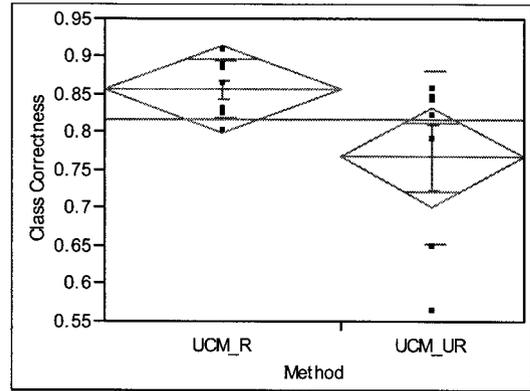


Figure 46 Distributions of CD Completeness - Lab 2.1

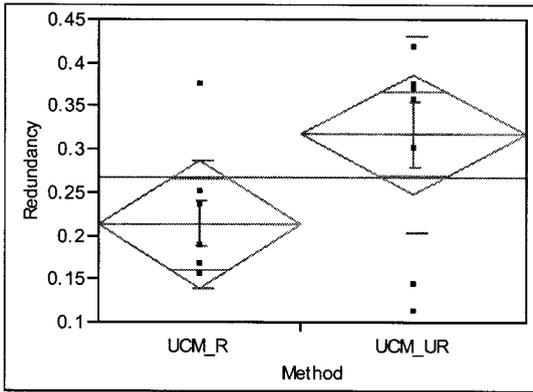


a) CPD

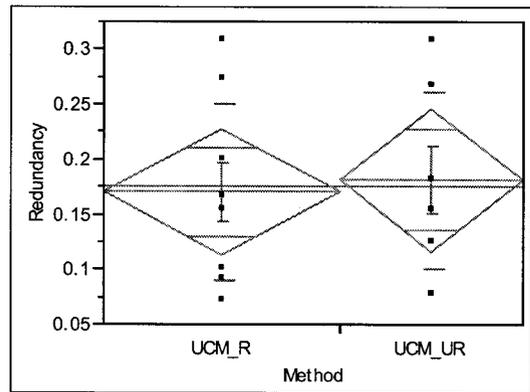


b) VS

Figure 47 Distributions of Class Correctness - Lab 2.1

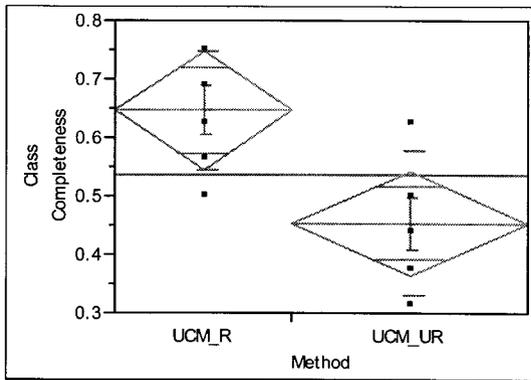


a) CPD

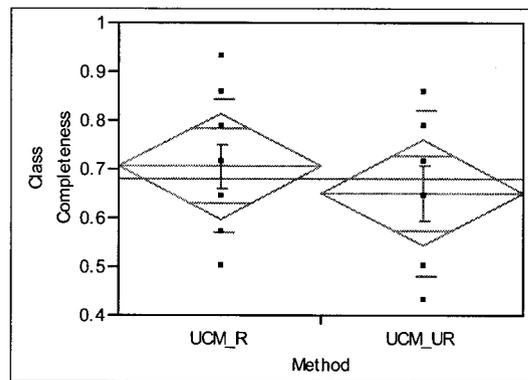


b) VS

Figure 48 Distributions of Redundancy - Lab 2.1

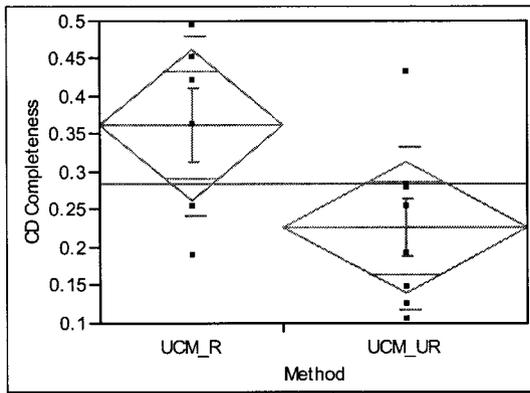


a) CPD

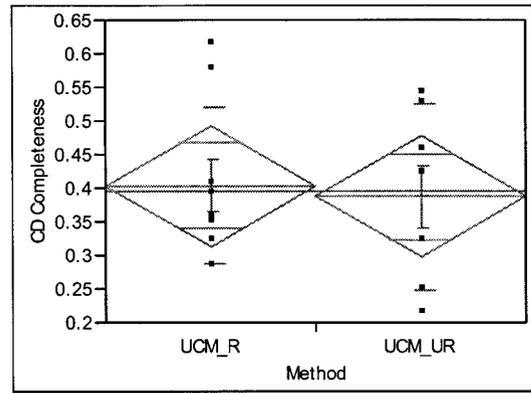


b) VS

Figure 49 Distributions of Class Completeness - Lab 2.3

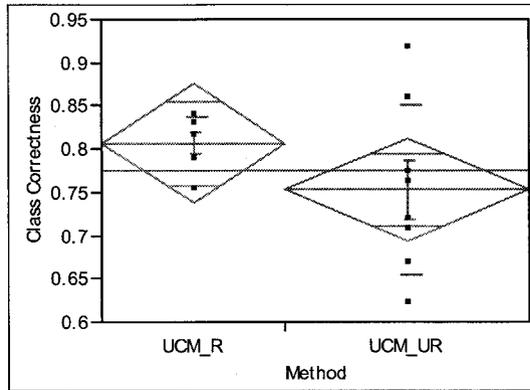


a) CPD

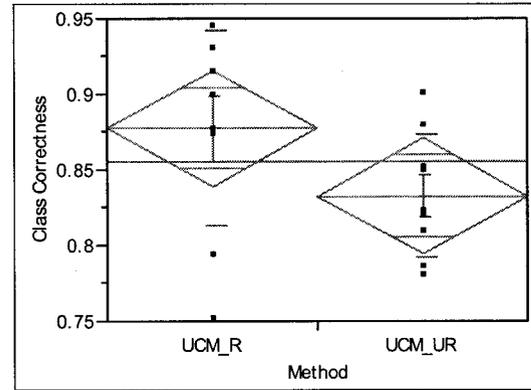


b) VS

Figure 50 Distributions of CD Completeness - Lab 2.3

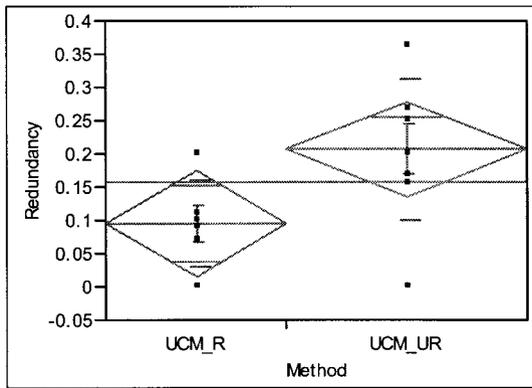


a) CPD

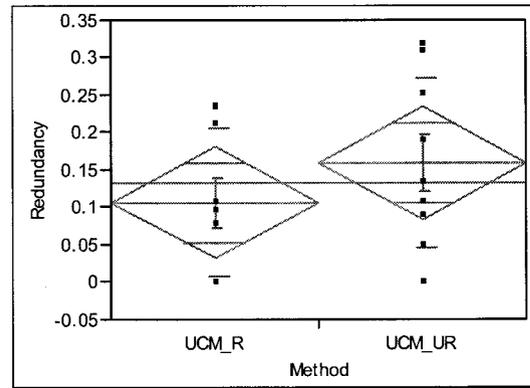


b) VS

Figure 51 Distributions of Class Correctness - Lab 2.3



a) CPD



b) VS

Figure 52 Distributions of Redundancy - Lab 2.3

Table 33 reports on the results of a two-tailed t -test to assess the statistical significance of differences of the four CD quality measures between UCM_R and UCM_UR. Equivalent non-parametric tests results (Wilcoxon rank sum test [108]) are also reported to confirm the validity of t -test results. Each row reports on each CD quality measure for each system (CPD or VS) or the two systems together (CPD+VS) for Experiment 1 (due to the small number of observations). Columns show the mean differences, degree of freedom (DF), the calculated t -value, and the corresponding p-value of the two-tailed t -test and corresponding probability for the non-parametric test.

Though the differences are not significant, the participants using UCM_R performed slightly better in terms of *Class Completeness*, *CD Completeness*, and *Redundancy* than those using UCM_UR. However, there is a statistically significant difference regarding *Class Correctness*: UCM_R yielded significantly higher quality class diagrams than UCM_UR.

In lab 2.1, UCM_R are better in terms of all four measures than UCM_UR. There are statistically significant differences regarding *Class Completeness* and *Class Correctness*. There is, however, a conflict between the results of the t -test and the non-parametric Wilcoxon test regarding *CD Completeness*. Various statistically-significant differences are observed for CPD only t -tests and no statistically-significant differences are observed for VS only t -tests, which indicates that it is necessary to perform a two-way ANOVA test to assess interaction effects between *System* (CPD vs. VS) and *Method* (UCM_R vs. UCM_UR).

In lab 2.3, again, UCM_R outperformed UCM_UR in terms of all the four measures. Statistically significant differences are observed in terms of *Class Completeness*, *Class Correctness*, and *Redundancy*, which is not consistent with the results of lab 2.1. This inconsistency is due perhaps to interaction effects between *Order* (2.1 vs. 2.3) and *Method* (UCM_R vs. UCM_UR). A two-way ANOVA test was performed to examine the impact of lab order on the dependent variable CD.

Table 33 Two tailed *t*-test and Wilcoxon test– CD

Exp	System	Measures	<i>t</i> -test				Wilcoxon test
			Mean difference (UCM R – UCM UR)	DF	t-value	p-value	Prob> Z
1	CPD+VS	Class Completeness	0.108	15	1.334	0.2022	0.1848
		CD Completeness	0.082	17	1.552	0.139	0.1564
		Class Correctness	0.138	16	2.281	0.037	0.0298
		Redundancy	-0.048	12	-0.792	0.4436	0.5372
2.1	CPD	Class Completeness	0.213	12	2.827	0.0157	0.0203
		CD Completeness	0.124	11	1.813	0.0961	0.1356
		Class Correctness	0.058	13	1.757	0.1022	0.1937
		Redundancy	-0.105	14	-2.269	0.0398	0.1472
	VS	Class Completeness	0.051	12	0.99	0.3411	0.4775
		CD Completeness	0.076	10	1.384	0.1956	0.39
		Class Correctness	0.09	7	1.986	0.0875	0.0567
		Redundancy	-0.011	13	0.27	0.7914	0.9576
	CPD+VS	Class Completeness	0.14	30	2.933	0.0064	0.0066
		CD Completeness	0.106	26	2.392	0.0242	0.0582
		Class Correctness	0.08	25	2.73	0.0114	0.021
		Redundancy	-0.068	26	-1.904	0.0682	0.2122
2.3	CPD	Class Completeness	0.193	12	3.188	0.008	0.0382
		CD Completeness	0.136	10	2.203	0.0514	0.0814
		Class Correctness	0.054	9	1.476	0.0175	0.22
		Redundancy	-0.111	12	-2.423	0.0326	0.0446
	VS	Class Completeness	0.056	15	0.768	0.454	0.532
		CD Completeness	0.017	16	0.277	0.7854	0.7904
		Class Correctness	0.045	13	1.765	0.1002	0.0934
		Redundancy	-0.053	16	-1.059	0.3056	0.2866
	CPD+VS	Class Completeness	0.124	29	2.326	0.0274	0.0344
		CD Completeness	0.076	30	1.634	0.1128	0.1214
		Class Correctness	0.054	30	2.112	0.0432	0.0892
		Redundancy	-0.08	30	-2.323	0.0272	0.0338

Interaction effects

The relationship between the dependent variable *CD* and *Method* may be affected by a number of interaction factors. A Two-way Analysis of Variance (ANOVA) was performed to test both the interaction between *Method* and *Order* and between *Method* and *System* for Experiment 2: Appendix M, Table 68.

For lab 2.1, the results show significant main effects of *Method* on all the four measures (in favor of UCM_R), significant main effects of *System* on *Class Correctness* and *Redundancy* (in favor of VS) and no significant interaction effects on any of the four

measures. For lab 2.3, the results show significant main effects of *Method* on *Class Completeness*, *Class Correctness*, and *Redundancy* (in favor of UCM_R). Significant main effects of *System* are observed in lab 2.3 for *Class Completeness*, *CD Completeness*, and *Class Correctness* (in favor of VS). No significant interaction effects are identified in lab 2.3 for any of the four measures. Least-square means plots are used to visualize the main and interaction effects of *Method* and *System* in lab 2.1 and lab 2.3: Appendix M, Figure 113 and Figure 114.

Lab order was the second factor for which we studied the interaction effect on *CD* to account for learning effects. We assume that participants would tend to perform better on lab 2.3 than lab 2.1, regardless of other factors. We performed a two-way ANOVA analysis to assess the impact of lab order on the four measures of CD: Appendix M, Table 69. Statistically significant differences are found for the main effects of *Order* in terms of *Class Correctness* and *Redundancy* (in favor of Lab 2.3), but no significant effect was observed for the interaction of *Method* and *Order*. We also observed that the participants performed better in Lab 2.3 than in Lab 2.1 in terms of all the four measures (least-square means plots are provided in Appendix M, Figure 115 for reference).

Summary

The participants applying UCM_R consistently performed better than the participants applying UCM_UR in terms of all four-quality measures in both Experiment 1 and Experiment 2. Statistically significant differences (in favor of UCM_R) are summarized in Table 34, when combining observations from the two systems to gain statistical power. ‘Yes’ means statistically significant difference is identified.

Table 34 Summary of *t*-test results of CD

Experiment	Class Completeness	CD Completeness	Class Correctness	Redundancy
1	No	No	Yes	No
2.1	Yes	Yes	Yes	No
2.3	Yes	No	Yes	Yes

We also observed that the participants in Experiment 2 outperformed the participants in Experiment 1 for both treatments with respect to *Class Completeness*, *CD Completeness*, and *Class Correctness*, but not *Redundancy*. Recall that the participants in Experiment 2 were given more time to derive class diagrams and we were expecting to obtain more complete and correct class diagrams and, as a result, more redundant classes (Section 12.2.4).

A two-way ANOVA was performed to test both the interaction between *Method* and *System* and between *Method* and *Order* for Experiment 2. No significant interaction effects were identified for any of the four measures. However, significant main effects of *System* and *Order* were found on various measures. Results are summarized in Table 35. Participants performed better on VS than on CPD perhaps because the use case model of VS contains five use cases while the use case model of CPD has six use cases, the latter thus requiring more time to understand.

Table 35 Summary of ANOVA tests - CD

Main effects	Experiment	Class Completeness	CD Completeness	Class Correctness	Redundancy
<i>System</i> (in favor of VS)	2.1	No	No	Yes	Yes
	2.3	Yes	Yes	Yes	No
<i>Order</i> (in favor of Lab 2.3)		No	No	Yes	Yes

12.3.2.2 Quality of analysis sequence diagram (SD)

In this section, we discuss the impact of *Method* (with two treatments *UCM_R* vs. *UCM_UR*) on the dependent variable *SD*, which is determined by different measures (Section 12.2.6.4). Next, we investigate the possible interactions between *Method* and possible interaction factors: *System* (*CPD* vs. *VS*) and *Order* (2.2 vs. 2.4). Descriptive statistics for *SD* are provided in Table 67, Appendix L for reference.

Recall, from Section 12.2.6.4, that a sequence diagram is evaluated in terms of a set of measures: *Completeness*, *Correctness*, *Redundancy*, and *BCE Consistency*. The completeness of a sequence diagram is calculated as the average of the completeness of

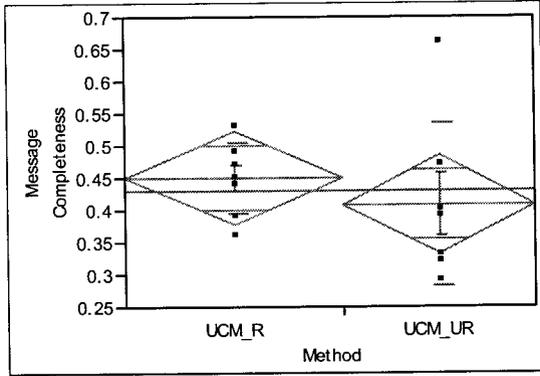
the messages, interaction uses, and combined fragments contained in the sequence diagram. The correctness of the sequence diagram is evaluated as the average of the correctness of the messages, interaction uses, and combined fragments of the sequence diagram. In the rest of the analysis, we report on a selected set of these measures: *Message Completeness*, *SD Completeness*, *Message Correctness*, *SD Correctness*, and *BCE Consistency*. The rationale is that *Message* is one of the most important elements of sequence diagrams and *SD Completeness* and *SD Correctness* indicate the overall completeness and correctness of a sequence diagram, including the completeness and correctness of its interaction uses and combined fragments. Therefore it is not necessary to report on the completeness and correctness of interaction uses and combined fragments separately. We also exclude *Redundancy* from the analysis of sequence diagrams as we observed very few sequence diagrams that contained redundant messages.

In the rest of the section, we first report univariate analysis results for *SD* on the selected five measures discussed above. Then interaction effects between *Method* and *System* and between *Method* and *Order* are discussed. Last, we summarize and discuss the analysis results.

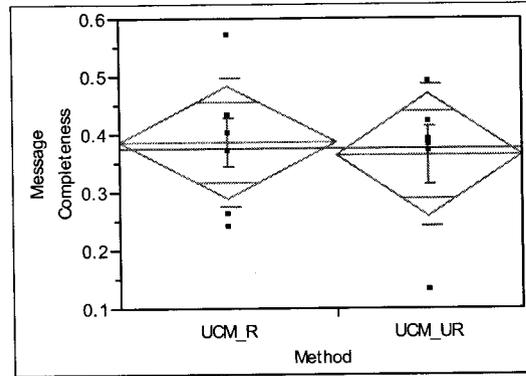
Univariate analysis

We performed univariate analysis to assess the isolated effect of *Method* on the five selected quality measures for *SD*: *Message Completeness*, *SD Completeness*, *Message Correctness*, *SD Correctness*, and *BCE Consistency*. Figure 53 to Figure 57 show mean diamonds for these measures in lab 2.2. Mean diamonds for all measures in lab 2.4 are shown in Figure 58 to Figure 62. In each of these figures, results are shown for each system to help comparisons.

We notice from these figures that UCM_R are better than UCM_UR in terms of most of the measures and for both systems in both lab 2.2 and lab 2.4.

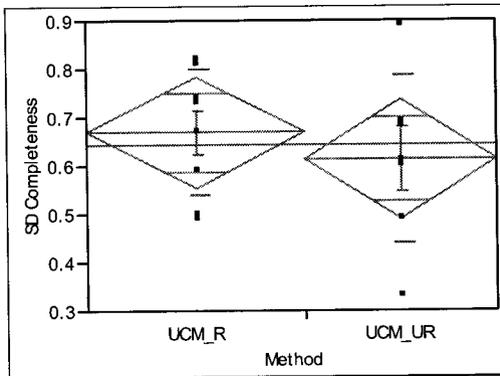


(a) CPD

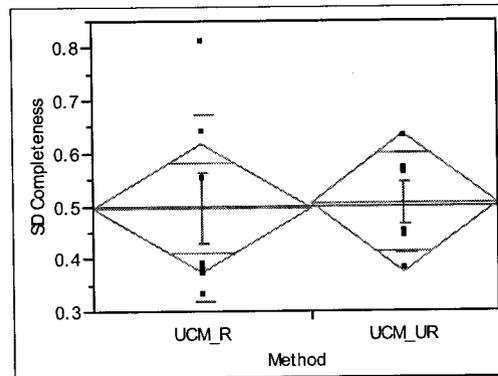


(b) VS

Figure 53 Distributions of Message Completeness - Lab 2.2

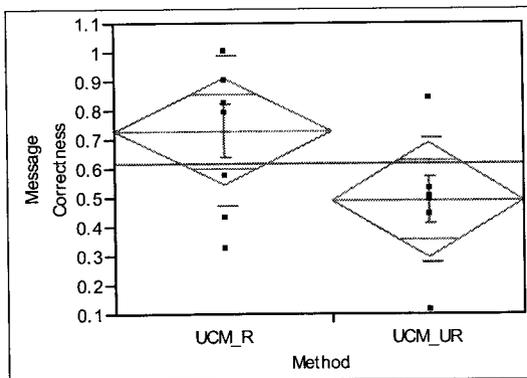


(a) CPD

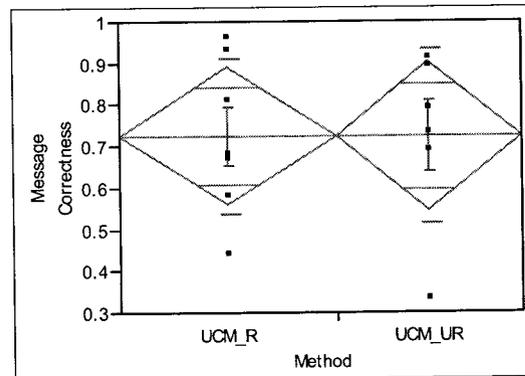


(b) VS

Figure 54 Distributions of SD Completeness - Lab 2.2

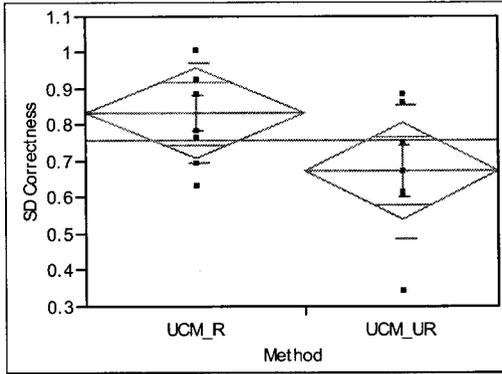


(a) CPD

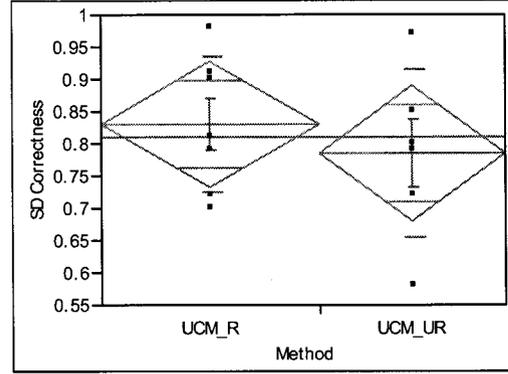


(b) VS

Figure 55 Distributions of Message Correctness - Lab 2.2

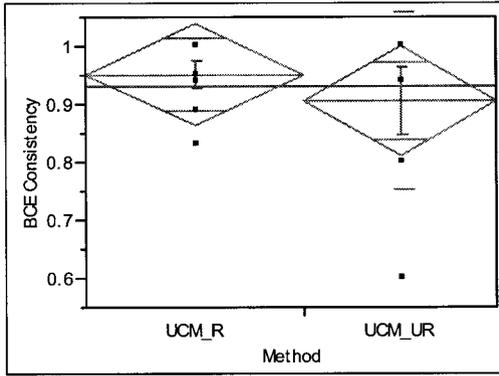


(a) CPD

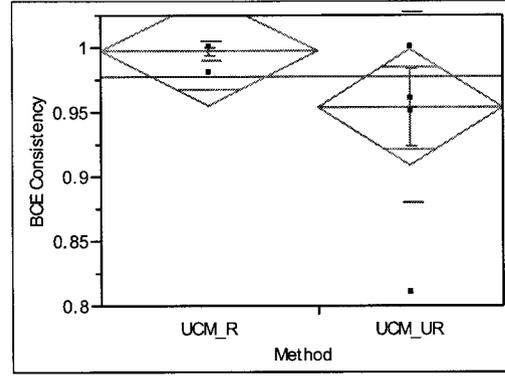


(b) VS

Figure 56 Distributions of SD Correctness - Lab 2.2

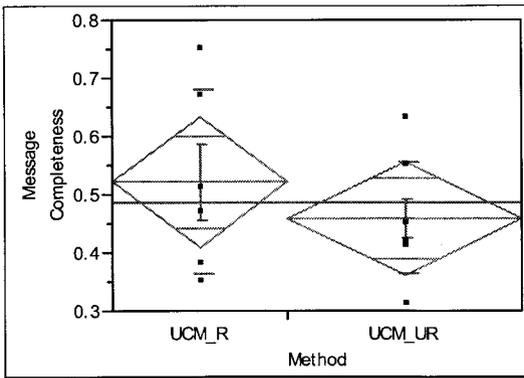


(a) CPD

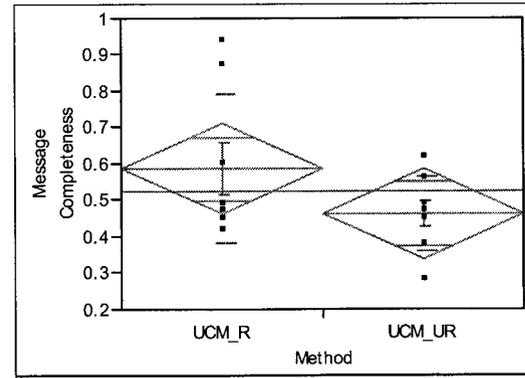


(b) VS

Figure 57 Distributions of BCE Consistency - Lab 2.2

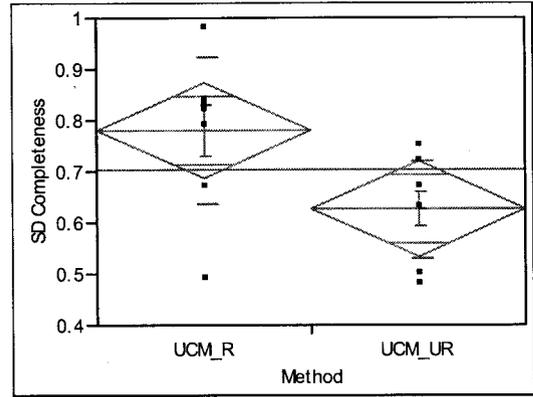
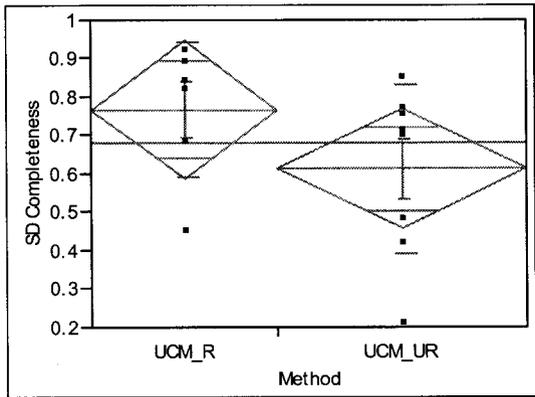


(a) CPD



(b) VS

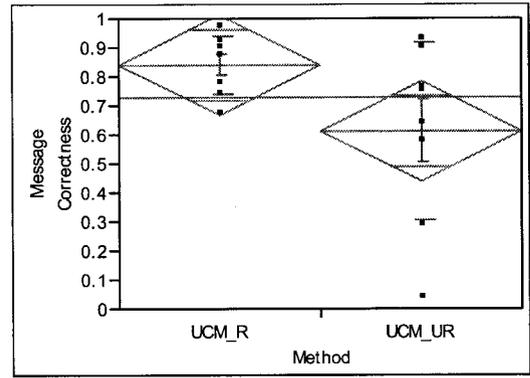
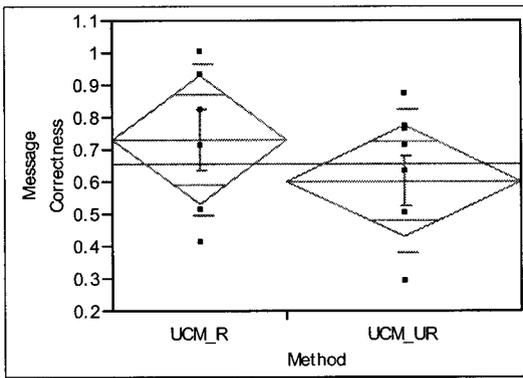
Figure 58 Distributions of Message Completeness - Lab 2.4



(a) CPD

(b) VS

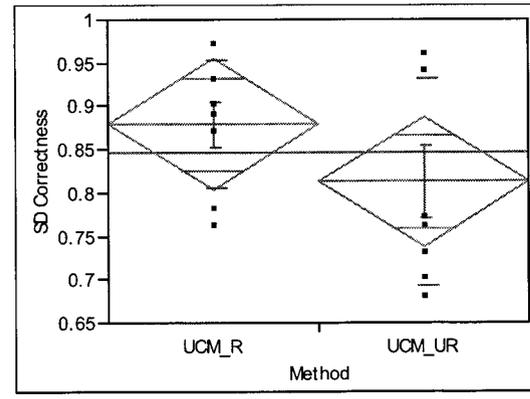
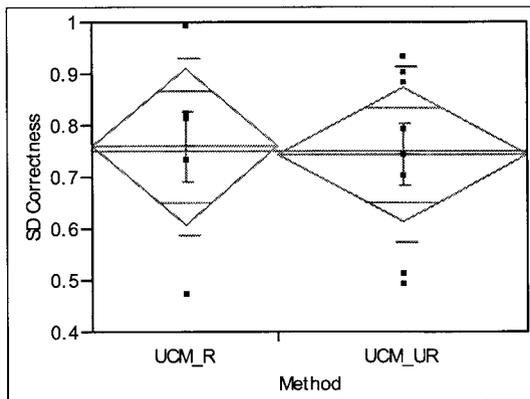
Figure 59 Distributions of SD Completeness - Lab 2.4



(a) CPD

(b) VS

Figure 60 Distributions of Message Correctness - Lab 2.4



(a) CPD

(b) VS

Figure 61 Distributions of SD Correctness - Lab 2.4

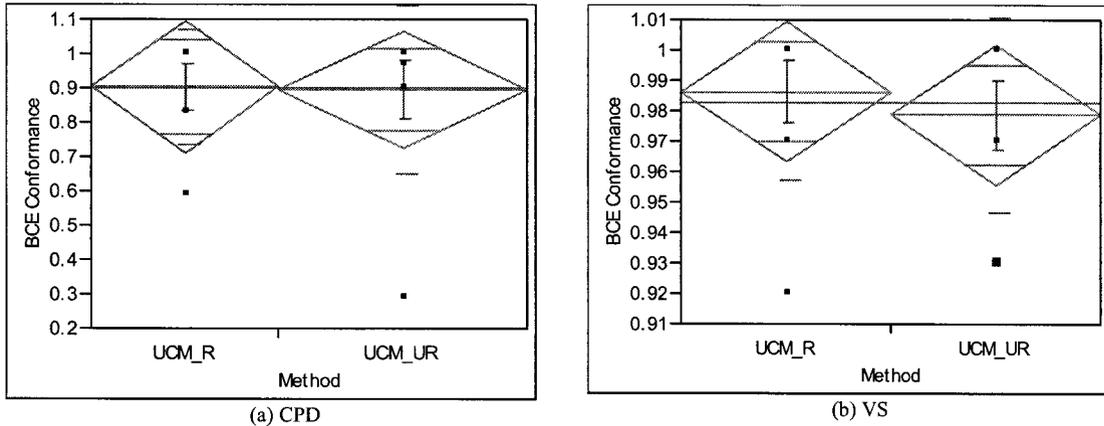


Figure 62 Distributions of BCE Consistency - Lab 2.4

Table 36 reports on the results of a two-tailed t -test for each system to assess the statistical significance of the difference for the five selected quality measures between UCM_R and UCM_UR. Non-parametric tests (Wilcoxon rank sum test [108]) were also performed to double-check the validity of t -test results.

In lab 2.2, there is no statistically significant difference observed. In lab 2.4, statistically significant differences regarding *SD Completeness* and *Message Correctness* are found based on the CPD+VS data points.

Table 36 One tailed t-test and Wilcoxon test– SD

Lab	System	t-test					Wilcoxon test
		Measures	Mean difference (UCM_R – UCM_UR)	DF	t-value	p-value	Prob> Z
2.2	CPD	Message Completeness	0.041	8	0.8061	0.4438	0.201
		SD Completeness	0.056	11	0.6955	0.5012	0.5624
		Message Correctness	0.237	13	1.9451	0.0738	0.2948
		SD Correctness	0.156	11	1.872	0.0878	0.0636
		BCE Consistency	0.046	8	0.7322	0.4856	0.9
	VS	Message Completeness	0.022	10	0.3419	0.7392	0.7746
		SD Completeness	-0.009	9	-0.1193	0.9076	0.5672
		Message Correctness	0.001	10	0.0085	0.9834	0.9432
		SD Correctness	0.045	10	0.6804	0.5124	0.6162
		BCE Consistency	0.044	5	1.45	0.2056	0.158
	CPD+VS	Message Completeness	0.032	22	0.792	0.437	0.2584
		SD Completeness	0.025	26	0.4094	0.6856	0.7952
		Message Correctness	0.128	25	1.4766	0.1524	0.1814
		SD Correctness	0.107	22	1.9275	0.0672	0.076
		BCE Consistency	0.045	16	1.247	0.2306	0.3742
2.4	CPD	Message Completeness	0.063	8	0.8596	0.416	0.5736
		SD Completeness	0.155	12	1.4738	0.1666	0.1752
		Message Correctness	0.128	11	1.0345	0.324	0.3006
		SD Correctness	0.016	11	0.1726	0.8662	1
		BCE Consistency	0.008	12	0.0751	0.9514	1
	VS	Message Completeness	0.124	10	1.5246	0.1572	0.3678
		SD Completeness	0.154	12	2.5168	0.0268	0.0204
		Message Correctness	0.229	8	2.0051	0.0778	0.1146
		SD Correctness	0.066	12	1.3277	0.2098	0.3436
		BCE Consistency	0.008	14	0.495	0.6286	0.7012
	CPD+VS	Message Completeness	0.098	19	1.798	0.088	0.1546
		SD Completeness	0.156	28	2.7023	0.0116	0.0076
		Message Correctness	0.186	26	2.3462	0.0268	0.0304
		SD Correctness	0.05	28	0.97	0.3404	0.3078
		BCE Consistency	0.014	26	0.259	0.7978	0.7433

Interaction effects

We performed a two-way Analysis of Variance (ANOVA) to assess possible interactions between *Method* and *Order* and between *Method* and *System*. ANOVA results showing the main and interaction effects of *Method* and *System* on the five *SD* quality measures are reported in Table 70 (Appendix M). For lab 2.2, the results show a significant main effect only for *System* on *SD Correctness*. For lab 2.4, the results show significant main effects of *Method* on *SD Completeness* and *Message Correctness*. No significant interaction effects are identified in both experiments.

We also use LSMeans plots (Appendix M, Figure 116 and Figure 117) to visualize the main and interaction effects of *Method* and *System* in lab 2.2 and lab 2.4. Results confirm that the participants with treatment UCM_R performed better than those with treatment UCM_UR. Unlike *CD* in lab 2.1 and lab 2.3 where the participants working on VS performed better than the participants working on CPD for all four quality measures (Section 12.2.6.4), we don't observe the same phenomenon for *SD* for lab 2.2 though we do in Lab 2.4 where significant differences between VS and CPD are observed for all the five measures.

To account for learning effects, the interaction effect of lab order on *SD* was also studied. We assume that participants would tend to perform better on lab 2.4 than lab 2.2, regardless of other factors. We performed a two-way ANOVA analysis to assess the impact of lab order on the five *SD* quality measures and the results are given in Table 71 (Appendix M). Corresponding LS Means plots are provided in Appendix M, Figure 118 to help visualizing the results. The results show a statistically significant impact of lab order through interactions with *Method* on *Message Completeness*, *SD Completeness*, *Message Correctness*, and *SD Correctness*, but not on *BCE Consistency*. The most plausible explanation is that participants improved their UML sequence diagram modeling skills from experience in lab 2.2 and therefore performed better in lab 2.4. This tentative explanation is confirmed by the participants when filling pre-lab questionnaires (Appendix L), reporting on how well they felt they mastered UML sequence diagram modeling.

Summary

The participants applying UCM_R consistently performed better than participants applying UCM_UR in terms of most of the five measures in Experiment 2. Statistically significant differences when combining observations from both systems are summarized in Table 37.

Table 37 Summary of *t*-test results of SD

Experiment	Message Completeness	SD Completeness	Message Correctness	SD Correctness	BCE Consistency
2.2	No	No	No	No	No
2.4	No	Yes	Yes	No	No

A two-way ANOVA was performed to test both the interaction between *Method* and *System* and between *Method* and *Order* for Experiment 2. No significant interaction effects were identified for any of the four measures. However, significant main effects of *System* and *Order* were found on various measures. The results are summarized in Table 35. Except for *BCE Consistency*, a statistically significant main effect of *Order* was identified for the other four measures in lab 2.4. This is because the participants, building on their experience in lab 2.2, performed much better in lab 2.4. This explanation is also confirmed from the responses of the participants to the pre-lab questionnaires in the two labs.

Table 38 Summary of ANOVA tests - SD

Main effects	Exp.	Message Completeness	SD Completeness	Message Correctness	SD Correctness	BCE Consistency
<i>System</i> (in favor of CPD)	2.2	No	Yes	No	No	No
	2.4	No	No	No	No	No
<i>Order</i> (in favor of lab 2.4)		Yes	Yes	Yes	Yes	No

12.3.2.3 Consistency between analysis class and sequence diagrams (CS)

The variable *CS* denotes the consistency between analysis class and sequence diagrams. It is the focus of one of the measures evaluating sequence diagrams: *SDCD Consistency* (Table 30). The descriptive statistics of the measure are presented in Table 39. Table 40 presents a summary of the statistical *t*-test results for *CS*. The results do not show statistically significant differences between the two treatments. This is reasonable because UCM_R does not in any way help designers keep consistency between analysis class and sequence diagrams. Non-parametric test results are not very different from the *t*-test results and are therefore not presented.

Table 39 Descriptive statistics of CS – lab 2.2 and lab 2.4

Methods	Lab 2.2		Lab 2.4	
	Mean	Size	Mean	Size
UCM_R	0.804	16	0.881	14
UCM_UR	0.812	14	0.848	16
All Methods	0.808	30	0.865	30

Table 40 One way t-test – CS – lab 2.2 and lab 2.4

Experiment	Mean difference (UCM_R – UCM_UR)	DF	t-value	p-value
2.2	-0.008	27	-0.149	0.4413
2.4	0.033	28	0.535	0.2986

12.3.2.4 Correctness of responses to the comprehension questionnaire

The dependent variable *QC* denotes the correctness of responses from a comprehension questionnaire (Sections 12.2.6.3 and 12.2.6.4). In this section, we report on two-tailed *t*-test results using the factor *Method*. The descriptive statistics of the measure are presented in Table 41 and the *t*-test results are given in Table 42.

The *t*-test result shows a significant difference between the two treatments in the expected direction for all the experiments, thus indicating an increased understanding due to restriction rules and the template. Non-parametric test results are not very different from the *t*-test results and are therefore not presented.

Table 41 Descriptive statistics of QC – Experiment 1, Experiment 2: lab 2.2, and lab 2.3

Methods	Experiment 1		Lab 2.2		Lab 2.3	
	Mean	Size	Mean	Size	Mean	Size
UCM_R	0.913	12	0.816	18	0.852	16
UCM_UR	0.527	8	0.545	15	0.494	18
All Methods	0.72	20	0.693	33	0.662	34

Table 42 t-test – QC – Experiment 1, Experiment 2: lab 2.2, and lab 2.3

Experiment	Mean difference (UCM_R – UCM_UR)	DF	t-value	p-value
1	0.387	8	5.189	<0.0008
2.2	0.271	29	4.4	<0.0002
2.3	0.356	26	8.4	<0.0002

12.4 Threats to Validity

Two main threats to external validity are related to our experiment, and are typical to controlled experiments in artificial settings and within time constraints: 1) are the subjects representative of software professionals? 2) Is the experiment material representative of industrial practice, in terms of the size of the systems we used?

Regarding issue 1), recall that in Task 1 the participants designed use case models by applying the restriction rules and the use case template we proposed. This task is usually performed by requirements engineers during the requirements elicitation phase of a typical software development lifecycle. Given the state of practice in most of the software industry, either participants or professional requirements engineers will likely require training. The participants of our experiment are 4th year software and computer engineering students who have received extensive training in use case modeling in previous courses. In addition, they were given a one-and-half-hour lecture and an assignment specifically focusing on how to apply the restriction rules and the use case template. In our context, the main difference between students and professional requirements engineers, is that the latter could have more experience on designing use case models, they could be more familiar with the domain, and thus we assume that they would probably apply more effectively the restriction rules and the use case template than students given the same amount of training and time to perform the task. Thus, we believe that professional requirements engineers would be able to further benefit from the restriction rules and template, and thus provide a more positive opinion on the rules' understandability, applicability, and restrictiveness. As for Task 2, the students derived analysis models from the use case models, with or without restrictions and template. This task is usually performed by system analysts. Again, our 4th year software and computer engineering students have received extensive training on software modeling with UML, through several courses, though they did not derive perfect class diagrams (specifically, poor use of class associations, though class correctness is good). This training is more than what we have observed in most software development environments.

As for issue 2) above, the scale of the systems is not likely to have a significant impact on the results of the experiment for Task 1. Indeed, this task does not require an overall understanding of the systems as the use case diagrams of the two systems were provided to the students as part of the experiment material. The students were only asked to write some UCSs by applying the restriction rules and the template. Due to time constraints (two three-hour laboratories), it was anyway not feasible to consider larger scale systems (with more UCSs) for Task 2.

Construct validity is related to our measurement instruments: the two comprehension questionnaires used respectively for the two tasks. The comprehension questionnaire for Task 1 (Section 12.2.5.1) was not involved in any comparison so it does not bear on construct validity. The questions of the comprehension questionnaire for Task 2 (Section 12.2.5.2) were designed to be answerable from the use case models with or without restrictions; therefore introducing no bias for any of the treatments. The same quality measures are used to measure the students class and sequence diagrams derived from the use case models with or without restrictions and therefore no bias is introduced as well when evaluating these diagrams. The only concern is that the textbook solutions used as reference models to evaluate aToucan generated models are not necessarily the only solutions: the results were obtained by comparing aToucan generated models to one possible solution for each case study problem, though different solution likely exist and could lead to different results; however these models are derived by authors of well-known textbooks (e.g., [15]) and we believe the quality of these models is sufficient to consider these models as reference models.

Three students presented problems related to internal validity. One of them missed the lab for Task 2 and had to perform the task at home; another spent 3 hours 40 mins on Task 2; the other produced a very incomplete result (no class diagram was derived). These three data points were excluded from the analysis. The participants belonging to different groups were monitored to ensure they would not access each other's documents during the entire lab duration. By doing so, we also limited the threat on the internal validity of the experiments.

12.5 Conclusion

Use case modeling is one of the most common practices for capturing functional requirements. However, use case specifications (UCSs) are essentially textual documents and therefore ambiguity is inevitably introduced. To facilitate the transition towards analysis models, the UCSs are expected to be the least ambiguous possible, especially when the goal is to provide automated support for the generation of analysis models. In this thesis, we propose a use case modeling approach, referred to as RUCM, which is composed of 26 well-defined restriction rules and a use case template. Its purpose is to restrict the use of natural language when users document UCSs in order to reduce ambiguity and also facilitate automated transition towards analysis models.

Two controlled experiments have been conducted, in the context of a fourth year software engineering course, to evaluate whether RUCM is easy to apply while developing use case models and whether it helps obtain higher quality analysis models. Each restriction rule is evaluated in terms of its understandability, applicability, restrictiveness, and error rate. The experiment results indicate that our 26 restriction rules are easy to apply and with focused training on the rules receiving higher error rates and appropriate tool support (e.g., enforcing the usage of keyword and the definition of compulsory alternative flows), error rates can be expected to further decrease. This forms our future work. Based on these results, we are therefore confident that trained engineers are capable of properly applying our restriction rules and template and obtain UCSs from which to derive analysis models.

Another goal of the controlled experiments was to evaluate whether RUCM helps derive higher quality analysis models, by comparing it to a common use case modeling approach that does not put restrictions on natural language. The quality of analysis class and sequence diagrams is evaluated from the viewpoints of their correctness, completeness, and redundancy. The results show that RUCM results in significant improvements regarding the class correctness of derived class diagrams consistently in all the experiments, but not their completeness and redundancy. The results also show that

RUCM leads to significant improvement on the diagram completeness and message correctness of derived sequence diagrams during the second lab of the second experiment. Furthermore, our approach resulted in a large improvement in term of comprehension of the use case models as measured by a carefully designed questionnaire.

To the best of our knowledge, this paper is the first controlled experiment that evaluates the applicability, both individually and as a whole, of restriction rules used to document UCSs and that also evaluates the impact of these rules and template on the quality of generated analysis class and sequence diagrams. The measures we have defined to characterize restriction rules and evaluate the quality of analysis class diagrams can be reused for similar experiments to be conducted in the future.

CHAPTER 13 EVALUATION OF CLASS DIAGRAMS

We used seven different software system descriptions to evaluate the generation of class diagrams of our approach. In the rest of this section, they are referred to as ATM (Automated Teller Machine), CPD (Car Part Dealer system), VS (Video Store system), CD (Cab Dispatching system), ARENA (tournament management system), Payroll, and Elevator. These systems come from different sources. ATM and Elevator are called Banking System and Elevator System in [39] respectively, and, for each of them, we have both the UCMod and analysis class diagram that we can use as a reference for evaluation purposes. Similarly, ARENA comes from [15], where a use case called Announce Tournament is described and the corresponding analysis class diagram is presented. Payroll comes from the IBM Rational course called “DEV496 Mastering IBM Rational Software Architect”: we have its UCMod and analysis model. All the above system models come from well-respected, expert sources, well-known book authors and professional courses. CPD, VS and CD were created (UCMod and analysis model) by Masters students well versed into UML modeling from our modeling courses, and subsequently reviewed and modified by the course instructors. Since the UCSs of these systems come from different sources, they were all re-written by applying RUCM. Note that CPD and VS were also used in the experiments to evaluate RUCM (Chapter 12): we therefore have a sample of 26 analysis models created by fully trained undergraduate students registered in a 4th year Software Engineering course at Carleton University, Ottawa, Canada, for the CPD and VS use case models. Table 43 summarizes some characteristics of each case study system: the numbers of use cases, simple sentences, and NPs in each UCMod (Columns 2-4), and the numbers of classes, associations, and generalization in each corresponding class diagram (Columns 5-7).

Table 43 Characteristics of each case study system – Class Diagram

Case studies	# of UC	# of simple sens.	# of NPs	# of classes	# of asso.	# of gen.
ATM	4	104	191	13	9	6
Elevator	2	40	91	10	10	0
CPD	13	177	347	16	11	4
VS	7	95	216	14	10	0
CD	10	90	226	6	5	2
ARENA	1	24	72	13	16	0
Payroll	13	202	396	12	6	6

Recall that aToucan can generate traceability links between a textual use case model and its automatically generated class diagram (through the intermediate model). However, we did not evaluate this aspect since the models we have for the seven systems do not contain this information.

In the rest of the section, we first discuss the design of the case studies (Section 13.1), followed by the presentation of the results (Section 13.2), a discussion (Section 13.3), and a performance analysis of our tool (Section 13.4). The automatically generated class diagrams of the seven case study systems are provided in Appendix F for reference.

13.1 Design of the study

Our goal is to investigate a number of research questions: How do analysis models created by aToucan compare to (1) models created by an expert (with ATM, Elevator, ARENA, and Payroll), (2) models from our modeling courses and reviewed over several years by course instructors (with CPD, VS and CD), and (3) models created by trained 4th year undergraduate students (with CPD and VS), respectively? To answer questions (1) and (2), we assess the Consistency and Completeness of the class diagram generated with aToucan by comparing it with the class diagram manually created by experts or Masters students from our modeling courses, depending on the model considered. These diagrams can be used as a basis for the assessment as they are considered (nearly) correct and complete, and therefore referred to as reference diagram. To address (3), we face the problem that both aToucan and 4th year students' diagrams are partially incomplete and incorrect, and we therefore have no means to compare them. Our strategy to answer (3) is to first perform a comparison between 4th year student diagrams and models from our

modeling courses (reference diagrams), thus obtaining a sample of consistency and completeness observations that are normally distributed. We then compare aToucan's diagrams with diagrams from our modeling courses, and get one value each for completeness and consistency. We can finally perform a one-sample statistical *t*-test to assess whether the average completeness and consistency of the 4th year student diagrams differ significantly from the average value obtained for the two aToucan diagrams (CPD and VS). We use a one-sample *t*-test as we only have one value of completeness and consistency for aToucan, which needs to be compared with the sample of diagrams from 4th year students.

13.2 Evaluation results

Table 44 lists our evaluation criteria results in separate columns for each system: Class consistency, Association consistency, Class diagram (CD) consistency (average class and association consistency), Class Completeness, Association completeness, Generalization completeness, and Class diagram Completeness (average of completeness of classes, associations and generalizations). Notice that "N/A" in Column "Gen" indicates that the reference class diagrams of those systems do not contain generalization relationships. The average measurement values over the different systems are provided in row 5 and row 9 to address research questions (1) and (2), respectively. As shown in Table 44, row 5, when compared with experts' models (1), on average aToucan achieves 92% class diagram consistency (97% and 85% class and association consistency), 51% class diagram completeness (73%, 34%, and 16% class, association, and generalization completeness). As shown in row 9, when compared with models from our modeling courses (2), aToucan consistently achieves high class diagram consistency (83%), low completeness for associations and generalizations, both of which contributing to dramatically lowering class diagram completeness to 54%, though class completeness is high (87%).

Table 45 presents a summary of the one-sample *t*-test results for all the evaluation criteria. The test sample is the set of the class diagrams created by 4th year engineering

students for the CPD and VS systems, which are evaluated (in terms of completeness and consistency) against the reference class diagrams from our modeling courses (Column 3). For instance, students' CPD and VS class diagrams show an average 0.79 class consistency. The tested value in a one sample *t*-test is the average, for a given evaluation criterion, of the two values for the CPD and VS class diagrams generated by aToucan, when compared against the reference models (Column 4). For example, aToucan class diagrams for CPD and VS have an average 0.86 class consistency when compared to reference (Modeling Courses) models. Note that tested values are obtained from Table 44: for instance, the class consistency value (Table 45, Column 4, Row 2: 0.86), is the average for CPD and VS in Table 44 (Column 4, Rows 6 and 7: 0.89, 0.83). The p-values in Table 45 show that aToucan significantly outperforms 4th year engineering students with respect to class consistency, class, association, and class diagram completeness.

Overall, the results show that aToucan performs well with respect to consistency and class completeness, and can therefore help generate a useful, initial analysis class diagram that can then be manually refined. We can observe that aToucan's results obtained with expert reference class diagrams are better than those obtained from our modeling courses. This is due to the fact that the former are of better quality, thus making it easier to achieve higher consistency. As a result, the values reported for experts are likely to be more credible.

Table 44 Evaluation results of all measures – aToucan (class diagram)

	Row#	Systems	Consistency			Completeness			
			Class	Asso	CD	Class	Asso.	Gen.	CD
Experts	1	ATM	0.97	0.67	0.82	0.69	0.44	0	0.34
	2	ARENA	0.96	1	0.98	0.54	0.06	N/A	0.57
	3	Payroll	0.96	0.75	0.86	1	0.67	0.33	0.67
	4	Elevator	1	1	1	0.7	0.2	N/A	0.45
	5	Avg.	0.97	0.85	0.92	0.73	0.34	0.16	0.51
Masters Students	6	CPD	0.89	0.71	0.8	0.69	1	0	0.56
	7	VS	0.83	1	0.92	0.93	0.7	N/A	0.82
	8	CD	0.83	0.71	0.77	1	0.6	0	0.53
	9	Avg.	0.85	0.80	0.83	0.87	0.76	0	0.54

Table 45 One sample *t*-test –4th students and aToucan (class diagram)

Measures		4 th yr Students Mean	aToucan Mean	DF	t-value	p-value
Consistency	Class	0.79	0.86	13	-2.15	0.026
	Asso.	0.98	0.86	13	10.47	1
	CD	0.89	0.86	13	1.13	0.86
Completeness	Class	0.48	0.81	13	-7.38	<.0001
	Asso.	0.11	0.85	13	-25.89	<.0001
	Gen.	0.09	0	13	1.44	0.9
	CD	0.26	0.69	13	-13.09	<.0001

13.3 Discussion

A careful investigation showed that the main reason leading to some classes not being generated by aToucan is that the corresponding information needed to generate these classes was not specified in UCMods, which somehow implies that the UCMods were not complete. For example, there are three classes missing in the class diagram generated by aToucan for ATM (Saving Account, Checking Account, and PIN Validation Transaction), when compared with the reference class diagram provided in [5]. A careful investigation found no related information provided by the UCMOD.

There are two reasons that explain why associations are hard to identify correctly: 1) some properties of associations (e.g., multiplicity and navigation) are very hard to obtain automatically through natural language parsing, and 2) missing classes lead to missing associations. It is also hard for aToucan to automatically generate generalizations just through parsing natural language. Manual refinement on the generated, initial class diagram is required to identify generalizations. It is also worth noticing that trained 4th year undergraduate students were not effective at identifying generalizations during our controlled experiment.

13.4 Performance analysis

aToucan was run on a laptop PC with Windows XP, Intel Core Duo 2.20 GHz and 2.19 GHz CPU, with 2 GB of RAM when the case studies were performed. We collected execution time for each case study and the data are reported in Table 46. The second row gives the execution time (seconds) of aToucan and the following two rows show the

number of simple sentences and NPs of the UCMod of each case study system, respectively. Execution time values are all in the order of a few minutes on a standard laptop and this suggests that even large UCMods should lead to reasonable processing times as long as the relationship between UCMod size and execution time is linear. Such linearity is suggested by the results presented in Table 46.

Table 46 Performance analysis results (class diagram)

	ATM	Elevator	CPD	VS	CD	ARENA	Payroll
Exe. Time (s)	88	37	329	187	139	31	385
# Simple Sen.	104	40	177	95	90	24	202
# of NPs	191	91	347	216	226	72	396

CHAPTER 14 EVALUATION OF SEQUENCE DIAGRAMS

We used six different software system descriptions to evaluate the generation of sequence diagrams of our approach. Five of them are the case study systems we used to evaluate generated class diagrams (Chapter 13): ATM, Elevator ARENA, CPD and VS. The same UCMods are used to generate both class and sequence diagrams. The other one is FRIEND (First Responder Interactive Emergency Navigational Database), which is reused from [15], where the use case *Report Emergency* and the corresponding analysis sequence diagram are described. Elevator has collaboration diagrams specified in [39]. Since UML 1.4 collaboration diagrams are equivalent to sequence diagrams, we mapped the ones for Elevator into their equivalent sequence diagrams, and used these as a reference for our evaluation purpose. Once again, we can consider these artifacts to be created by experts (renowned textbook authors) and therefore we can use them as a reference for evaluation purposes. As discussed in Chapter 13, the UCMods of CPD and VS were created by Masters students, refined by instructors for our modeling courses, and the sequence diagrams were automatically generated by aToucan. These two systems are used to compare generated sequence diagrams with those manually created by trained 4th year undergraduate students in laboratory conditions. Table 47 summarizes some characteristics of each case study system: use cases, the number of messages, lifelines, interaction use (IU), and combined fragments (CF) of each reference sequence diagram. As for class diagrams and for the same reason, we did not evaluate the traceability links generated by aToucan.

Table 47 Characteristics of each case study system - sequence diagram

Case studies	Use cases	# of messages	# of lifelines	# of IU	# of CF
ATM	Validate PIN	15	7	0	0
	Withdraw Funds	27	9	0	0
Elevator	Select Destination	6	6	0	0
	Request Elevator	7	6	0	0
ARENA	Announce Tournament	30	15	0	0
FRIEND	Report Emergency	20	11	0	0
CPD	Create Customer Order	13	10	1	1
	Order Part	14	6	2	2
	Insufficient Stock	16	8	1	1
VS	Rent Video	27	11	4	3
	Check Database	33	12	4	7

In the rest of the section, we first discuss the design of the case studies (Section 14.1), followed by the discussion of evaluation analysis and results (Section 14.2). The sequence diagrams automatically generated by aToucan for the ATM system case study are provided in Appendix G.

14.1 Design of study

Similar to what we did for class diagrams, we aim to answer two research questions: How do sequence diagrams created by aToucan compare to 1) sequence diagrams created by an expert (ATM, Elevator, ARENA, and FRIEND), and 2) sequence diagrams created by trained 4th year undergraduate students (CPD and VS)? To answer the first question, we compare sequence diagrams generated with aToucan with sequence diagrams manually devised by textbook authors using the evaluation criteria in Section 12.2.6.4. As for class diagrams, these diagrams were used as a basis for the assessment as they are considered correct and complete, and therefore used as reference diagrams. To answer the second question, we compare the sequence diagrams manually created by 4th year undergraduate students in course labs with the ones automatically generated by aToucan.

14.2 Evaluation results and analysis

Results from comparing the sequence diagrams generated by aToucan with experts' solutions are provided in Table 48 for each of our evaluation criteria: *Message Consistency*, *Message Completeness*, and *Message Redundancy*. Notice that the sequence

diagrams created by textbook authors do not fully conform to UML 2 sequence diagrams (i.e., no interaction uses and combined fragments were derived). These diagrams also do not comply with the BCE design principle (Section 12.2.6.4). For these four case studies, we therefore cannot assess the diagrams generated by aToucan against experts' solutions for the following measures: *IU Consistency*, *IU Completeness*, *CF Consistency*, *CF Completeness*, and *BCE Consistency*. As a result not all measures defined in 12.2.6.4 can be used for comparison. Besides, textbook sequence diagrams are not always consistent with their corresponding class diagrams and, therefore, aToucan cannot be evaluated against the expert-derived sequence diagrams in terms of *SDCD Consistency*. aToucan generated sequence diagrams have the following characteristics: 1) they fully comply with the BCE principle (100% *BCE Consistency*) and 2) they are fully consistent with their corresponding class diagrams (100% *SDCD Consistency*).

As shown in the last row of Table 48, on average, when compared with experts, aToucan achieved very high message consistency (91%) and completeness (97%), 100% message ordering consistency, and near 0% SD redundancy. aToucan achieved relatively low message consistency for two use cases of the Elevator system: 75% and 83% for use cases *Select Destination* and *Request Elevator*, respectively. A careful investigation showed that the collaboration diagrams in the source textbook do not fully conform to their UCMods. The collaboration diagrams are very coarse-grained and some steps described in the UCMods are not realized in the collaboration diagrams. However, the sequence diagrams automatically generated by aToucan fully correspond to the UCMods and are therefore not fully consistent with the textbook collaboration diagrams.

Table 48 Evaluation results of aToucan against experts – sequence diagram

Systems	Use cases	Message Consistency	Message Completeness	Message Ordering Consistency	SD Redundancy
ARENA	Announce Tournament	0.99	0.87	1.00	0.03
FRIEND	Report Emergency	0.99	0.93	1.00	0.00
Elevator	Select Destination	0.75	1.00	1.00	0.00
	Request Elevator	0.83	1.00	1.00	0.00
ATM	Validate PIN	0.95	1.00	1.00	0.00
	Withdraw Funds	0.96	1.00	1.00	0.00
Average		0.91	0.97	1.00	0.01

Table 49 presents a summary of the comparison of aToucan-generated sequence diagrams with the ones from trained 4th year undergraduate students for two systems: CPD and VS. During a three-hour experiment, the students were asked to derive sequence diagrams for the CPD and VS system use cases of Table 47. Some students were not able to derive all required sequence diagrams and we decided to discard their observations in order to obtain comparable data, i.e., data based on the same sets of sequence diagrams. Eventually we obtained samples of 17 and 13 observations for CPD and VS, respectively. Table 49 shows the mean values of these two samples with respect to different criteria. For example, the messages of the sequence diagrams manually derived by students for the systems CPD and VS are on average 78% and 73% consistent with the messages of the sequence diagrams automatically generated by aToucan, respectively. Recall (from Section 12.2.6.4) that the measure SD Consistency is computed as the consistency average for Message, IU, CF, and Message Ordering. From Table 49, we can observe that the students only achieved 52% and 45% Message Completeness for CPD and VS, respectively. In other words, on average, nearly half of the messages of aToucan-generated sequence diagrams were not created by students. Due to the clear specifications of sequential steps in flows of events in the use cases, students achieved 99% Message Ordering Consistency for both systems. For a similar reason, the sequence diagrams manually derived by students contain zero message redundancy. Results in Table 49 clearly suggest that trained students are unlikely to perform as well as aToucan when deriving sequence diagrams from use case specifications.

Table 49 Evaluation results of 4th students against aToucan – sequence diagram

	Consistency					Completeness				Redun dancy	BCE Consistency	SDCD Consistency
	Msg	IU	CF	Msg Ordering	SD	Msg	IU	CF	SD			
CPD	0.78	0.11	0.62	0.99	0.86	0.52	0.91	0.75	0.72	0	0.97	0.85
VS	0.73	1	0.59	0.99	0.8	0.45	0.8	0.62	0.56	0	0.95	0.88

CHAPTER 15 EVALUATION OF ACTIVITY DIAGRAMS

In this section, we discuss how we validated our approach (Section 15.1) and also compare our approach with three commercial tools (Section 15.2). In Section 15.3, we discuss the related work and the comparison between our approach and the related work.

15.1 Validation procedure and summary of results

We used five different software system descriptions (18 use cases altogether) to assess our approach. They are from different sources: three are from textbooks and two were created by Masters students. Since the UCSs of these systems come from different sources, they were re-written by applying RUCM.

The goal of our validation was two-fold: (1) To assess whether our transformation rules are complete: do they accommodate all UCSs in our case studies? (2) To determine whether our transformation rules lead to activity diagrams that are syntactically and semantically correct. Syntactic correctness means that a generated activity diagram conforms to the UML 2.0 activity diagram notation. Semantic correctness means that a generated activity diagram correctly represents its UCS; all the steps described in the flows of events of the UCS are correctly transformed by following the transformation rules and no redundant model elements are generated. In order to check correctness and completeness, the validation procedure is as follows.

1. Given a UCMOD in RUMC as input, aToucan automatically generates an activity diagram for each UCS of the UCMOD.
2. For each UCS, we check whether each step of the flows of events and the precondition and postconditions have been properly transformed.
3. We check whether each generated activity diagram is syntactically and semantically correct.
4. We check whether the data flow information (input and output pins) attached to each activity diagram is properly generated.

Following the above procedure, for all 18 use cases, we achieved 100% completeness and correctness with aToucan, and 100% of the traceability links were also correctly established. Regarding the completeness and correctness of data flow information attached to each activity diagram, aToucan was not able to generate input and output pins for some actions. First, transformation rules D2.1-2.3 (used to generate data flow information) rely on package `SentenceStructure` of `UCMeta`. Recall that a NL parser is used in our approach to parse each textual sentence and the parsing result is transformed into instances of model elements (e.g., `Object`) of package `SentenceStructure` (Section 4.2.4). The NL parser has limitations and cannot always produce a correct result. Therefore the instances of the model elements of package `SentenceStructure` do not always correctly correspond to their textual sentences. Second, each generated pin is typed to a class of the class diagram; however the automatically generated class diagram is not 100% correct and complete (see Chapter 7 for details), again partly because of limitations of the NL parser. As a result, it is possible that there is no matching class found for an element of a sentence (such as an object—recall rule B2 in Section 9.2) and therefore no pin is generated. Besides, whether data flow information can be deemed correct also depends on what it is used for. For example, if it is used to automatically generate test cases, manual refinement of the automatically generated data flow information is absolutely required. Therefore, here, we don't evaluate the correctness of generated data flow information but its completeness, measured by the ratio of occurrences of missing pins over the total number of instances of `CallOperationAction` in an activity diagram. Table 50 summarizes the completeness of data flow information of every use case of all the five case study systems. Results show that the average completeness of data flow information across all the five case study systems is 85%.

Table 50 Completeness of data flow information

Case studies	Use cases	Total # of CallOperationAction	Occurrences of missing pins
ATM	Withdraw Funds	22	1
	Query Account	8	1
	Transfer Fund	17	2
	Validate PIN	16	3
Elevator	Select Destination	13	2
	Request Elevator	14	2
CPD	Order Part	8	1
	Insufficient Stock	4	1
	Create Vendor Order	6	2
	Create Preventive Order	5	1
	Create Customer Order	5	0
	Complete Pending Order	6	1
VS	Rent Video	13	2
	Reserve Video	12	1
	Return Video	9	2
	Check Database	15	2
	Video Overdue	9	1
ARENA	Announce Tournament	24	4
Total		206	29
%		29/206 = 85%	

15.2 Comparison with three commercial tools

Visual Paradigm [104], Ravenflow [85], and CaseComplete [17] are commercial tools that can automatically transform requirements into UML activity diagrams. We tested them by using the use case of Table 10 since it contains three different types of alternative flows, concurrency sentences, and validation sentences. Various features of UCSs are therefore considered and this use case can be considered complete in terms of UCS and generated activity diagram features. The UCS was rewritten according to the format requirements of each tool. The details of the re-written UCSs and automatically generated activity diagrams are provided in Appendix G for reference. In the rest of the section, we summarize their main differences.

1. Visual Paradigm and CaseComplete can transform the flows of events of a use case into an activity diagram. Each flow of events needs to be structured using a simple use case template (basic flow and its extensions). Ravenflow does not require a use case template, but a set of writing guidelines are proposed (not enforced by the tool though) to guide users to write sentences that can be correctly parsed by the tool. For

example, "if...then.... Otherwise,..." is suggested to write a conditional sentence. A "!" at the end of a sentence indicates the termination of a flow. Since Ravenflow does not require UCSs be structured, alternative flows may be very hard to describe in unstructured sentences. aToucan is based on RUCM (a use case template and a set of restriction rules), which have been experimentally evaluated to be easy to apply (Chapter 12).

2. None of the three commercial tools can generate forks and joins because concurrency sentences are not recognized. Our approach is based on RUCM, which specifies the keyword MEANWHILE (Section 3.3) to help users specify concurrency sentences. Therefore, aToucan can generate a fork, a join, and a set of parallel sentences between the fork and the join (Section 9.2) for each concurrency sentence. Visual Paradigm and CaseComplete do not support swimlanes. Both our approach and Ravenflow support swimlanes—one swimlane per actor and one swimlane for the system—but have different mechanisms to identify actors. Ravenflow relies on Natural Language Processing (NLP) techniques to identify possible actors to generate corresponding swimlanes, which means that the tool might falsely identify actors. Recall that Ravenflow does not have a use case template to structure use case steps. However, our tool is based on RUCM, and primary and secondary actors of each use case are clearly specified in each UCS. Also thanks to RUCM, aToucan can also automatically transform global alternative flows (Section 9.2). It is very hard (if not impossible) to find an alternative way to specify global alternative flows in the requirements format required/enforced by the three commercial tools.
3. None of the three commercial tools can support include and extend use case relationships because they can only transform a single use case instead of a use case model (UCMod). aToucan takes a UCMod as input and use case relationships are naturally supported.
4. Visual Paradigm and CaseComplete cannot generate any data flow information since

they do not use any NLP technique. Ravenflow can generate data flow information but to a quite limited extent: it generates data flow only when the data is manipulated by two swimlanes, as indicated in the writing guidelines provided along with the tool. This means that Ravenflow cannot derive data flow information from sentences with transaction types `InternalTransaction` and `Validation`. aToucan does not have such a limitation.

15.3 Related Work and Comparison

We conducted a systematic literature review (Chapter 12) on transformations of textual requirements into analysis models, including class, sequence, and activity diagrams. The review identified 20 primary studies (16 approaches) based on a carefully designed paper selection procedure in scientific journals and conferences from 1996 to 2008 and Software Engineering textbooks. The method proposed here is based on the results of this review, with a particular focus on automatically deriving activity diagrams from UCMods. There also exists several literature works recently published that were therefore not included in our systematic literature review. In this section, we evaluate our approach by comparing it with these existing literature works and also three existing commercial tools: Visual Paradigm for UML [104], Ravenflow [85], and CaseComplete [17]. We define a set of evaluation criteria for comparison, which are in part from the system review we conducted (Chapter 12):

- 1) **Requirements:** We need to know the requirements format (e.g., formalized use cases) required by a specific approach so that we can assess how difficult it is to document requirements.
- 2) **NLP:** We need to be aware of whether or not any NLP techniques are applied. We can then assess whether or not certain features (e.g., automatically derive swimlanes) can be supported by the approach.
- 3) **Automation:** This criterion evaluates whether a transformation is automated, automatable, semi-automated, or manual. An approach is automated if it has been fully implemented. If a transformation algorithm is proposed in a paper, then we assess whether we deem the description to be sufficient to implement it, and if this is

the case, the transformation approach is deemed automatable. In some cases, a transformation is semi-automated because user interventions are required. Last, some approaches are entirely manual.

- 4) **Traceability:** We check whether traceability links between requirements and analysis model elements are established when a transformation is performed.
- 5) **Objective:** The original objective of each approach can help us understand their limitations and motivate our work.
- 6) **Activity diagram:** We evaluate the activity diagrams that each approach is able to derive from requirements with respect to the following four aspects: 1) their types (standard, extended, or non-standard notation), 2) important model elements that are expected to be generated (e.g., swimlanes), 3) whether include and extend relationships of use cases are supported, and 4) whether data flow information can be generated. Activity diagrams conforming to the UML specification [75] are standard activity diagrams; extended activity diagrams are those based on a profile of the UML specification; non-standard activity diagrams do not conform to the UML specification.

The evaluation results are summarized in Table 51 and Table 52. The first columns of these two tables show the approaches we evaluated. The first four rows are the approaches proposed in existing research works; the following three rows are the selected commercial tools; the last row is our approach: aToucan. The rest of the columns are arranged according to the evaluation criteria.

Table 51 Evaluation summary (part I)

Approach	Requirements	NLP	Automation	Traceability	Objective
[42]	Formalized UCs	No	Automated	No	Visualize use cases and facilitate test generation
[48]	Unstructured requirements	Yes	Automatable	No	Complement analysis models
[33]	Unstructured requirements	Yes	Semi-automated	No	Complement analysis models
[69]	Exceptional UCs	--	Manually	No	Formalize UCs to reduce ambiguities
[104]	Flows of events of UCs	No	Automated	Yes	Visualize flows of events of UCs
[85]	Restricted sequential steps	Yes	Automated	Yes	Visualize textual sequential steps
[17]	Flows of events of UCs	No	Automated	No	Visualize flows of events of UCs
aToucan	RUCM models	Yes	Automated	Yes	All of the above

Table 52 Evaluation summary (part II)

Approach	Activity Diagram						
	Type	Swimlane	Fork and join	Decision node	Global alternative flows	Include or extend	Data flow
[42]	Standard	Yes	No	Yes	No	No	No
[48]	Extended	No	No	Yes	No	--	No
[33]	Standard	No	Yes	No	No	--	No
[69]	Extended	--	--	--	--	--	--
[104]	Standard	No	No	Yes	No	No	No
[85]	Standard	Yes	No	Yes	No	--	Yes
[17]	Non-standard	No	No	No	No	No	No
aToucan	Standard	Yes	Yes	Yes	Yes	Yes	Yes

As shown in Column 2, Table 51, one approach [42] formalizes use cases as instances of a metamodel, similarly to aToucan. However, the metamodel instance has to be manually provided by the user directly, instead of being transformed automatically from another (more simple) representation (RUCM), thereby leading to substantial user effort. Two approaches ([48] and [33]) take unstructured requirements (plain text) as inputs to derive activity diagrams. Both are not fully automated. An approach is proposed in [69] to manually transform exceptional use cases into extended activity diagrams. Special stereotypes (e.g., <<failure>> and <<handler>>) are introduced to specify exceptional handling concepts. Visual Paradigm [104] and CaseComplete [17] can automatically

transform flows of events of a use case into an activity diagram. Both tools require a similar and simple use case template to structure flows of events. Ravenflow [85] can automatically visualize a set of sequential and textual steps into an activity diagram and no structured format (e.g., template) is needed to document these steps. Ravenflow however suggests users follow a set of writing principles, some of which are very similar to the restriction rules of RUCM used in aToucan. Because Ravenflow does not rely on a use case template, it becomes very difficult to specify alternative flows in a use case and their interactions with the basic flow.

As shown in Column 3, Table 51, except for one manual approach, three existing approaches rely on NLP techniques. The approach proposed in [42] does not apply any NLP techniques because it requires formalized use cases as its inputs. Visual Paradigm [104] does not rely on any NLP technique; therefore it cannot automatically generate swimlanes, forks and joins, for instance. Four existing approaches are automated (Column 4) and only two commercial tools have traceability capability (Column 5).

The objectives of the existing approaches are different, as shown in Column 6, Table 51. The approach proposed in [42] aims to visualize use cases and therefore automated test generation can be facilitated. Visualizing use cases or their scenarios for the purpose of better understanding and analyzing them is a common practice [10, 38] and the idea of activity diagram-based test generation is also promoted in [19, 71]. Both the approaches proposed in [48] and [33] can generate analysis models including class and activity diagrams. Generated activity diagrams, as part of the generated analysis models, model dynamic behavior of a system. The approach proposed in [69] however simply uses activity diagrams as a means to formalize textual use cases. All three commercial tools visualize either flows of events of use cases (i.e., [17, 104]) or sequential textual steps (i.e., [85]) in activity diagrams for the purpose of helping users to construct and understand requirements. Our approach however applies to any of these objectives.

The approaches proposed in [48, 69] cannot generate standard UML activity diagrams (Column 2, Table 52). Mustafiz et al. [69] propose an approach to manually transform

exceptional use case (with elements that allow the modeling of system behavior in exceptional situations) into activity diagrams extended by specific stereotypes. Ilieva and Ormandjieva [48] propose an automatable approach to transform requirements into extended activity diagrams—activity diagrams integrated with the concepts of actors, business rules, and messages. As shown in Columns 3-6, except for the manual approach, swimlanes are only supported by two approaches: one of them [42] requires formalized use cases as input and the other [85] relies on NLP techniques to automatically identify swimlanes (similarly to aToucan); only one approach [33] supports forks and joins but it is semi-automated; decision nodes are supported by most of the non-manual approaches; Global alternative flows are not supported by any of the existing approaches. The approaches that are not manual and take use cases as inputs, do not support include and extend relationships of use cases. Ravenflow is the only existing approach that can generate data flow information (similarly to aToucan), and we have discussed differences in Section 15.1.

To compare with these existing approaches, our approach can automatically transform each use case of a UCMOD into two types of standard UML 2.0 activity diagrams while fully supporting traceability. Additionally, as part of the functionality of aToucan, automatically generated activity diagrams are naturally consistent with other UML analysis model diagrams such as sequence diagrams and horizontal traceability (across different diagrams) can then be supported. Swimlanes, decision nodes, forks and joins, include and extend relationships, and data flow information are all supported by our approach. Besides, thanks to RUCM, our approach can also transform global alternative flows.

CHAPTER 16 INDUSTRIAL CASE STUDIES

We conducted two industrial case studies to evaluate (1) the applicability of RUCM on the requirements of real industrial systems and (2) to check whether aToucan, for various diagrams, generates many incorrect model elements. One case study is a communication system (video conferencing system) developed by Tandberg [100]. The core functionality of this typical video conferencing system is sending and receiving multimedia streams. In this case study, we experimented with eight use cases of the system. In the other case study, we modeled part of the functionality of one type of FMC's subsea systems [36] as a RUCM model with seven use cases. FMC's subsea systems are integrated control systems used to explore, drill and develop offshore oil and gas fields.

RUCM was successfully applied to model 15 use cases from the two case studies, with the help of two domain experts. We did not meet any difficulty in applying RUCM and it took on average three hours to complete the UCMods of each system. This suggests that the constructs in RUCM were adequate to express the two systems' requirements.

RUCM models were input to aToucan to generate analysis class and sequence diagrams. Since we did not have reference class or sequence diagrams as in the artificial studies used in the previous sections, we needed to perform evaluation by relying on human expertise. The generated diagrams were evaluated by the same domain experts that helped with the RUCM models. By using the same experts, we could make sure that any discrepancy in the generated analysis class and sequence diagrams was due to the transformation itself, and not possible differences of perspectives among experts. The following questions have been designed and provided to the domain experts to collect their feedback on the generated analysis models.

1. Do the classes in the generated class diagram correspond to domain concepts?
2. Do the attributes and operations of each class in the generated class diagram reflect the properties and responsibilities of the domain concept captured by the class?

3. Are all the domain concepts captured in the UCSs represented in the class diagram?
4. Do the associations in the generated class diagram express meaningful connections between domain concepts?
5. Do the messages in each generated sequence diagram conform to the system events of each use case?
6. Does the ordering of the messages of each generated sequence diagram correctly reflect the ordering of the steps of the flow of events of each UCS?
7. Are the generated sequence diagrams consistent with the class diagram according to the defined consistency rules in Section 12.2.6.4?

For each of the questions above, experts were asked to list all model element instances for which the question's answer was negative, e.g, classes not matching a domain concept for question 1. Based on the experts' responses, the number of such negative instances are reported in Table 54 for each question. For example, for Q1, the number of classes in the generated class diagram that do not correspond to any domain concept is, for each of the two systems, three and five, respectively.

As shown in the table, for Q1 (Column 2), three (five) classes generated for the Tandberg (FMC technologies) case study were considered by the domain experts not to conform to any domain concept, which represent 16% (23%) of the total numbers of generated classes in the class diagrams, respectively. In terms of Q2, almost all the attributes and operations of each class in the generated class diagrams were considered reflecting the properties and responsibilities of the domain concept captured by the class, as shown in Column 3. Only one operation of a class in the generated class diagram for the Tandberg case study is not related to any responsibility of the domain concept captured by the class, which represents only 1 out of 25 of the operations of all the generated classes. For Q3, as shown in Column 4, all the domain concepts captured in the UCSs are represented in the class diagrams, for both of the two case studies. There is however one class in the class diagram generated for the Tandberg case study, which was considered to be redundant. Regarding Q4, only one association in the class diagram generated for the

Tandberg case study (1 out of 16) has a wrong multiplicity, according to the domain expert, as shown in Column 5. It is interesting to notice that all the generated messages were considered to conform to the system events of the use cases (for Q5, Column 6). Moreover, the automatically generated sequence diagrams are fully consistent with the automatically generated class diagrams for both case studies (for Q7, Column 8). As shown in Column 7, there are only two places where the ordering of the steps of the generated sequence diagram for the FMCtechnology case study was considered wrong.

So based on two industrial case studies, we can conclude that, overall, the vast majority of model elements generated by aToucan are correct. This implies that such models would, in practice, be good initial models to refine and augment to converge towards a correct and complete analysis model.

An inspection of the resulting analysis models also showed to fully comply with the UML syntax, a problem frequently encountered in industrial models. Though this should not come as a surprise, it is a clear advantage of automated model generation. Regarding performance, it took aToucan less than five minutes to generate the class diagrams including respectively 33 classes (FMC) and 19 classes (Tandberg), and nine sequence diagrams with 43 Messages and nine CombinedFragments (FMC) and seven sequence diagrams with 39 Messages and ten CombinedFragments (Tandberg), as shown in Table 53.

Table 53 Characteristics of the generated analysis models of each industrial case study system

	Tandberg	FMCtechnologies
# of use cases	8	7
# of classes	19	22
# of attributes of all classes	20	20
# of operations of all classes	25	22
# of associations	16	48
# of generalizations	8	7
# of messages	39	43
# of combined fragments	10	9

Table 54 Responses from domain experts on the evaluation questions

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Tandberg	3	1	0	1	0	0	0
FMCtechnologies	5	0	0	0	0	2	0

CHAPTER 17 SUMMARY

The summary presented in this chapter includes the conclusion (Section 17.1) and future research directions (Section 17.2).

17.1 Conclusion

Deriving UML analysis models from use case requirements is an important step of model-driven development; however this step has not received enough attention and limited automated support has been proposed. This mostly stems from the fact that requirements (e.g., use case specifications) are essentially textual documents and tend to be more difficult to analyze than other software artifacts. In this thesis, we propose a method and a tool (aToucan) to automatically generate an initial UML analysis model from a use case model (UCMod) documented by using a use case modeling approach (RUCM) specifically designed to facilitate this automated transition.

Developing aToucan was motivated by a systematic literature review (Chapter 12), from which we observed that existing approaches either impose unacceptable constraints to document requirements or are not able to efficiently and (semi-)automatically generate a reasonably complete (i.e., static and dynamic aspects) and consistent analysis model. The systematic review also suggested that a promising path of investigation was to devise an approach to automatically and efficiently transform a UCMod, described using natural language restrictions, into a complete, correct and consistent UML model comprising both structural and behavioral aspects.

Our approach and tool (aToucan) complies with this objective and features the following advantages.

1. RUCM, the use case modeling approach used to document requirements, has been empirically shown not to be too constraining and resulted in significant improvements over the use of a standard UCMod template.

2. aToucan is fully automated and our prototype tool is built on the top of a set of advanced natural language and model engineering technologies: It is easy to extend, modify, and integrate with open source (e.g., [27]) or commercial tools (e.g., [46]).
3. UCMeta, the intermediate (MOF-based) model used by aToucan, is the most complete use case metamodel to the best of our knowledge. It covers the UML::Usecases package, the use case template of RUCM, and relevant natural language concepts.
4. aToucan can automatically establish traceability links between requirements and analysis model elements when transformations are performed, which is paramount to support software evolution.
5. The transformation rules of aToucan are well structured and precisely specified, so that they are easy to extend and modify.
6. An extensive evaluation has been conducted to evaluate aToucan, including two industrial case studies.

Seven case studies have been performed to evaluate class diagrams and the results show that aToucan can achieve high consistency and class completeness when compared with a reference class diagram derived by experts. aToucan also significantly outperforms trained, 4th year software engineering students in terms of the consistency and the completeness of classes and associations in analysis class diagrams. Performance analysis results of aToucan shows that its execution time is linearly dependent on the number of simple sentences contained in a UCMod and remains within the range of a few minutes on a low end laptop computer, thus suggesting that the approach is scalable. We also compared the sequence diagrams generated by our tool to the ones devised by experts and trained 4th year undergraduate students. Results show that the sequence diagrams automatically generated by our tool are highly consistent with the ones devised by experts and are also very complete. Results also show that the automatically generated diagrams are far more complete than the ones manually created by trained students.

Two industrial case studies from two different domains (communication systems and integrated control systems) were conducted to evaluate aToucan in realistic conditions.

The results demonstrate that RUCM is applicable on real requirements and that aToucan was able to generate meaningful analysis models within a few minutes.

17.2 Future research directions

aToucan currently can generate class, sequence and activity diagrams. State machine diagrams, as another important type of behavior diagrams of UML, should also be automatically generated by aToucan. The motivation is that there is a need to have an automated approach to generate UML state machines from textual UCMOD, in order to facilitate Model-Based Testing (MBT) [64], more specifically state machine-based test case generation.

Tool support is needed to help requirements engineers to construct RUCM models, by for example automatically or semi-automatically enforcing the application of the RUCM restrictions (Section 3.3). For example, rule R1 states that “The subject of a sentence in basic and alternative flows should be the system or an actor” (Table 7). This restriction can be automatically checked. If it is violated, a warning message would then be provided to tool users. Regarding enforcing the application of the RUCM keywords (e.g., VALIDATES THAT), the tool could provide a list of these keywords for the tool users to select when they write a use case specification. In general, the tool should be able to facilitate RUCM model specification and identify flawed requirements.

Using a glossary or domain model to specify domain concepts is a common practice in industry. Adding such sources of information in RUCM might help improve the quality of the aToucan-generated analysis models. The impact of these two artifacts on the quality of derived analysis model, especially class diagrams, should be investigated.

REFERENCES

- [1] Parsing, <http://en.wikipedia.org/wiki/Parsing> (Last accessed April 2008)
- [2] Sentence, <http://en.wikipedia.org/wiki/Sentence> (Last accessed April 2008)
- [3] Abbott R. J., "Program design by informal English descriptions," *Com. ACM*, vol. 26 (11), pp. 882-894, 1983.
- [4] Achour C. B., Rolland C., Maiden N. A. M. and Souveyet C., "Guiding use case authoring: Results of an empirical study," *Proc. RE'99*, pp. 36-43, 1999.
- [5] Ambriola V. and Gervasi V., "On the Systematic Analysis of Natural Language Requirements with CIRCE," *Automated Software Engineering*, vol. 13 (1), pp. 107-167, 2006.
- [6] Amyot D. and Andrade R., "Use Case Maps for the capture and Validation of Distributed Systems Requirements," *Proc. ISRE'99*, 1999.
- [7] Anda B., Sjoberg D. and Jorgensen M., "Quality and understandability of use case models," *Proc. ECOOP 2001*, pp. 402-428, 2001.
- [8] ATL, Atlas Transformation Language (ATL), <http://www.eclipse.org/m2m/atl/> (Last accessed March 2008)
- [9] Basili V. R., Caldiera G. and Rombach H. D., "The goal question metric approach," *Encyclopedia of Software Engineering*, vol. 1, pp. 528-532, 1994.
- [10] Berenbach B., Inc S. C. R. and Princeton N. J., "The evaluation of large, complex UML analysis and design models," *Proc. ICSE*, 2004.
- [11] Bittner K. and Spence I., *Use Case Modeling*, Addison-Wesley Boston, 2002.
- [12] Borgo S., Gangemi A., Guarino N., Masolo C. and Oltramari A., WonderWeb Deliverable D15 Ontology RoadMap, <http://wonderweb.semanticweb.org/deliverables/documents/D15.pdf>
- [13] Brown E. K. and Miller J. E., *Syntax: a linguistic introduction to sentence structure*, Routledge, 1992.
- [14] Bruegge B. and Dutoit A. H., *Object-Oriented Software Engineering Using UML, Patterns, and Java*, Prentice Hall, 2nd Edition, 2004.

- [15] Bruegge B. and Dutoit A. H., *Object-Oriented Software Engineering Using UML, Patterns, and Java*, Prentice Hall, 3rd Edition, 2009.
- [16] Capuchino A. M., Juristo N. and Van de Riet R. P., "Formal justification in object-oriented modelling: A linguistic approach," *Data & Knowledge Engineering*, vol. 33 (1), pp. 25-47, 2000.
- [17] CaseComplete, <http://www.casecomplete.com/>
- [18] Champeaux D. D., *Object-Oriented Development Process and Metrics*, Prentice Hall, 1 edition (September 9, 1996) Edition, 1996.
- [19] Chen T. Y., Tang S. F., Poon P. L. and Tse T. H., "Identification of categories and choices in activity diagrams," *Proc. QSIC 2005*, pp. 55-63, 2005.
- [20] Christiansen H., Have C. T. and Tveitane K., "From use cases to UML class diagram using logic grammars and constraints," *Proc. Recent Advances in Natural Language Processing*, pp. 128-132, 2007.
- [21] Cockburn A., *Writing effective use cases*, Addison-Wesley Boston, 2001.
- [22] Cox K., *Heuristics for use case descriptions*, PhD Thesis, Bournemouth University, UK, 2002
- [23] Czarnecki K. and Helsen S., "Classification of Model Transformation Approaches," *Proc. OOPSLA Workshop on Generative Techniques in the Context of the MDA*, 2003.
- [24] Czarnecki K. and Helsen S., "Feature-based survey of model transformation approaches," *IBM Systems Journal*, vol. 45 (3), pp. 621-645, 2006.
- [25] Diaz I., Losavio F., Matteo A. and Pastor O., "A specification pattern for use cases," *INFORM MANAG*, vol. 41 (8), pp. 961-975, 2004.
- [26] Diaz I., Pastor O. and Matteo A., "Modeling Interactions using Role-Driven Patterns," *Proc. IEEE International Conference on Requirements Engineering*, pp. 209-220, 2005.
- [27] Eclipse Foundation, Eclipse Modeling Framework, <http://www.eclipse.org/modeling/emf/> (Last accessed April 2010)
- [28] Eclipse Foundation, Eclipse Technology Project, www.eclipse.org/technology/index.php (Last accessed November 2008)
- [29] Eclipse Foundation, UML2: EMF-Based UML 2.0 Metamodel Implementation, www.eclipse.org/uml2/ (Last accessed November 2008)

- [30] Egyed A., Biffi S., Heindl M. and Grunbacher P., "Determining the cost-quality trade-off for automated software traceability," *Proc. ASE2005*.
- [31] Feijs L. M. G., "Natural language and message sequence chart representation of use cases," *IST*, vol. 42 (9), pp. 633-647, 2000.
- [32] Feijs L. M. G., "Natural language and message sequence chart representation of use cases," *Information and Software Technology*, vol. 42 (9), pp. 633-647, 2000.
- [33] Fliedl G., Kop C., Mayr H. C., Salbrechter A., Vöhringer J., Weber G. and Winkler C., "Deriving static and dynamic concepts from software requirements using sophisticated tagging," *Data Knowl. Eng.*, vol. 61 (3), pp. 433-448, 2007.
- [34] Fliedl G., Kop C., Mayr H. C., Salbrechter A., Vöhringer J., Weber G. and Winkler C., "Deriving static and dynamic concepts from software requirements using sophisticated tagging," *Data & Knowledge Engineering*, vol. 61 (3), pp. 433-448, 2007.
- [35] Fliedl G., Mayerthaler W., Winkler C., Kop C. and Mayr H. C., "Enhancing requirements engineering by natural language based conceptual predesign," *Proc. IEEE International Conference on Systems, Man, and Cybernetics*, 5, pp. 778-783, 1999.
- [36] FMCtechnologies, <http://www.fmctechnologies.com/>
- [37] Fortuna M. H., Werner C. M. L. and Borges M. R. S., "Info Cases: Integrating Use Cases and Domain Models," *Proc. RE '08*.
- [38] Fowler M., *UML distilled: a brief guide to the standard object modeling language*, Addison-Wesley, 2003.
- [39] Goma H., *Designing Concurrent, Distributed, and Real-Time Applications with UML*, Addison-Wesley, 2000.
- [40] Greenbaum S., *The Oxford English Grammar*, Oxford University Press, 1996.
- [41] Grubacher P., Egyed A. and Medvidovic N., "Reconciling software requirements and architectures with intermediate models," *Software and Systems Modeling*, vol. 3 (3), pp. 235-253, 2004.
- [42] Gutiérrez J. J., Clémentine N., Escalona M. J., Mejías M. and Ramos I. M., "Visualization of Use Cases through Automatically Generated Activity Diagrams," *Proc. MODELS2008*.
- [43] Harmain H. M. and Gaizauskas R., "CM-Builder: A natural language-based CASE tool for object-oriented analysis," *Automated Software Engineering*, vol. 10 (2), pp. 157-181, 2003.

- [44] IBM-Rational, "Rational Unified Process," 2008.
- [45] IBM, IBM Model Transformation Framework
- [46] IBM, Rational Software Architect
- [47] IBM Model Transformation Framework, IBM, <http://www.alphaworks.ibm.com/tech/mtf> (Last accessed November 2008)
- [48] Ilieva M. G. and Ormandjieva O., "Models Derived from Automatically Analyzed Textual User Requirements," *Proc. on Software Engineering Research, Management and Applications*, 2006.
- [49] Insfrán E., Pastor O. and Wieringa R., "Requirements Engineering-Based Conceptual Modelling," *Requirements Engineering*, vol. 7 (2), pp. 61-72, 2002.
- [50] Ivar J., "Object-oriented development in an industrial environment," *Proc. OOPSLA*, 1987.
- [51] Jacobson I., Booch G. and Rumbaugh J., *The Unified Software Development Process*, Addison-Wesley, 1999.
- [52] Jacobson I., Christerson M., Jonsson P. and Overgaard G., *Object-oriented software engineering: a use case driven approach*, Addison-Wesley, 1992.
- [53] Jouault F. and Kurtev I., "Transforming models with ATL," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 3844, pp. 128, 2006.
- [54] Kealey J. and Amyot D., "Towards the Automated Conversion of Natural-Language use Cases to Graphical use Case Maps," in *CCECE*, 2006, pp. 2377-2380.
- [55] Kermeta, Kermeta metaprogramming environment, <http://www.kermeta.org/> (Last accessed April 2010)
- [56] Kitchenham B. A., "Guidelines for performing systematic literature reviews in software engineering," EBSE Technical Report EBSE-2007-001, 2007.
- [57] Kleppe A., Warmer J. and Bast W., *MDA Explained - The Model Driven Architecture: Practice and Promise*, Addison-Wesley, 2003.
- [58] Kruchten P., *The Rational Unified Process: An Introduction*, Addison-Wesley, 2003.
- [59] Kulak D. and Guiney E., *Use cases: requirements in context*, ACM Press, 2000.

- [60] Lamsweerde A. V., *Requirements engineering: from systems goals to UML models to software specifications*, Wiley, 2009.
- [61] Larman C., *Applying UML and Patterns*, Prentice-Hall, 3rd Edition, 2004.
- [62] Lethbridge T. C. and Laganieri R., *Object-Oriented Software Engineering: Practical software development using UML and Java*, McGraw-Hill Education, 2001.
- [63] Liu D., *Automating Transition from Use Cases to Class Model*, Thesis, University of Calgary, Department of Electrical and Computer Engineering, 2003
- [64] M. U. and B. L., *Practical Model-Based Testing: A Tools Approach*, Morgan-Kaufmann, 2006.
- [65] Manning C. D. and Schutze H., *Foundations of statistical natural language processing*, MIT Press, 1999.
- [66] Matthew B. D., Lori A. C., Jamieson M. C. and Gleb N., "Flow analysis for verifying properties of concurrent software systems," *TOSEM*, vol. 13 (4), pp. 359-430, 2004.
- [67] Mayr H. C. and Kop C., "A User Centered Approach to Requirements Modeling," *LNI; Vol. 12*, pp. 75-86, 2002.
- [68] Mich L., "NL-OOPS: from natural language to object oriented requirements using the natural language processing system LOLITA," *Natural Language Engineering*, vol. 2 (02), pp. 161-187, 1996.
- [69] Mustafiz S., Kienzle J. and Vangheluwe H., "Model transformation of dependability-focused requirements models," *Proc. ICSE Workshop on Modeling in Software Engineering*, 2009.
- [70] Nakatani T., Urai T., Ohmura S. and Tamai T., "A Requirements Description Metamodel for Use Cases," *Proc. Asia-Pacific on Software Engineering Conference* 2001.
- [71] Nebut C., Fleurey F., Le Traon Y. and Jezequel J. M., "Automatic test generation: A use case driven approach," *IEEE TSE*, vol. 32 (3), pp. 140-155, 2006.
- [72] OMG, "Model Driven Architecture (MDA)."
- [73] OMG, "MOF 2.0 Core Specification (formal/2006-01-01)."
- [74] OMG, "MOF Query/Views/Transformations V1.0."
- [75] OMG, "UML 2.2 Superstructure Specification (formal/2009-02-04)."

- [76] OMG, "OCL 2.0 Specification," Object Management Group, Final Adopted Specification ptc/03-10-14, 2003.
- [77] OMG, "UML 2.0 Superstructure Specification," Object Management Group, <http://www.omg.org/spec/UML/2.0/>, 2005.
- [78] OMG, "MOF 2.0 Core Final Adopted Specification," 2008.
- [79] Oppenheim A. N., *Questionnaire design, interviewing, and attitude measurement*, Pinter Pub Ltd, 1992.
- [80] Overmyer S. P., Benoit L. and Owen R., "Conceptual modeling through linguistic analysis using LIDA," in *ICSE'01*, 2001, pp. 401-410.
- [81] Pender T., *UML Bible*, Wiley, 2003.
- [82] Phalp K. T., Vincent J. and Cox K., "Assessing the quality of use case descriptions," *Software Quality Journal*, vol. 15 (1), pp. 69-97, 2007.
- [83] Phalp K. T., Vincent J. and Cox K., "Improving the quality of use case descriptions: empirical assessment of writing guidelines," *Software Quality Journal*, vol. 15 (4), pp. 383-399, 2007.
- [84] Pressman R. S., *software engineering: a practitioner's approach*, McGraw-Hill, 6th Edition, 2005.
- [85] RAVENFLOW, <http://www.ravenflow.com/>
- [86] Richards D., "Merging individual conceptual models of requirements," *Requirements Engineering*, vol. 8 (4), pp. 195-205, 2003.
- [87] Ryser J. and Glinz M., "SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test," Technical Report. University of Zurich, 2003.
- [88] Salbrechter A., Mayr H. C. and Kop C., "Mapping Pre-designed Business Process Models to UML In: Hamza MH (Hrsg.)," *Proc. IASTED International Conference on Software Engineering and Applications Cambridge USA*, 2004.
- [89] Samarasinghe N. and Somé S., "Generating a Domain Model from a Use Case Model," *Proc. Intelligent and Adaptive Systems and Software Engineering*, 2005.
- [90] Schneider G. and Winters J. P., *Applying use cases: a practical guide*, Object Technology, Addison-Wesley, 1998.

- [91] Sen S., Baudry B. and Mottu J. M., "On Combining Multi-formalism Knowledge to Select Models for Model Transformation Testing," *Proc. ICST 2008*, pp. 328-337, 2008.
- [92] Śmiałek M., Bojarski J., Nowakowski W., Ambroziewicz A. and Straszak T., "Complementary Use Case Scenario Representations Based on Domain Vocabularies," *Proc. MoDELS*, 2007.
- [93] Śmiałek M., Kalnins A., Kalnina E., Ambroziewicz A., Straszak T. and Wolter K., "Comprehensive System for Systematic Case-Driven Software Reuse," *Proc. SOFSEM 2010*, 5901/2010, pp. 697-708, 2010.
- [94] Somé S. S., "An approach for the synthesis of State transition graphs from Use Cases," *Proc. Software Engineering Research and Practise*, 1, pp. 456-462, 2003.
- [95] Somé S. S., "Supporting use case based requirements engineering," *Information and Software Technology*, vol. 48 (1), pp. 43-58, 2006.
- [96] Sommerville I., *Software Engineering*, Addison Wesley, 7th Edition, 2004.
- [97] Subramaniam K., Far B. H. and Eberlein A., "Automating the transition from stakeholders' requests to use cases in OOAD," *Proc. Canadian Conference on Electrical and Computer Engineering*, 1, pp. 0515-0518, 2004.
- [98] Subramaniam K., Liu D., Far B. H. and Eberlein A., "UCDA: Use Case Driven Development Assistant Tool for Class Model Generation," *Proc. SEKE'04*, 2004.
- [99] Tan H. B. K., Yang Y. and Bian L., "Systematic transformation of functional analysis model into OO design and implementation," *IEEE TSE*, vol. 32 (2), pp. 111-135, 2006.
- [100] Tandberg, <http://www.tandberg.com/index2.jsp2010>)
- [101] The Stanford Natural Language Processing Group: The Stanford Parser version 1.6.
- [102] Toro A. D., Jimenez B. B., Cortes A. R. and Bonilla M. T., "A Requirements Elicitation Approach Based in Templates and Patterns," *Proc. Workshop de Engenharia de Requisitos. Buenos Aires, Argentina*, 1999.
- [103] Vincenzo G. and Didar Z., "Reasoning about inconsistencies in natural language requirements," *TOSEM*, vol. 14 (3), pp. 277-330, 2005.
- [104] Visual Paradigm International, Visual Paradigm for UML

- [105] Waheed T., Iqbal M. Z. Z. and Malik Z. I., "Data Flow Analysis of UML Action Semantics for Executable Models," *Proc. ECMDA-FA*, 2008.
- [106] Wahono R. S. and Far B. H., "A framework for object identification and refinement process in object-oriented analysis and design," *Proc. Cognitive Informatics*, pp. 351-360, 2002.
- [107] Wahono R. S. and Far B. H., "A framework for object identification and refinement process in object-oriented analysis and design," *Proc. First IEEE International Conference on Cognitive Informatics*, pp. 351-360, 2002.
- [108] Wohlin C. and Wesslen A., *Experimentation in Software Engineering: An Introduction*, Springer, 2000.
- [109] Young R. R., *Effective Requirements Practices*, Addison-Wesley, 2001.

Appendix A RUCM Restriction rules

Table 55 Restrictions (part 1)

#	Description	Example		Location to apply
		RIGHT	WRONG	
1	The subjects of sentences in basic and alternative flows should be the system or actors.	<i>The system ejects the ATM card</i>	<i>The card has been ejected</i>	Apply to action steps of basic and alternative flows of events. Don't apply to conditions: • IF conditions • ELSEIF conditions • VALIDATES THAT conditions • DO-UNTIL conditions
2	Describe the flow of events sequentially.	1. The system dispenses the cash amount. 2. The system ejects the ATM card.	1. The system ejects the ATM card. 2. The system dispenses the cash amount.	
3	Actor-to-actor interactions are not allowed.	The customer inserts the ATM card into the card reader.	The customer gives the teller the ATM card.	
4	Describe one action per sentence. (Avoid compound predicates.)	the system cancels the transaction MEANWHILE the system ejects the card. or 1. the system cancels the transaction. 2. the system ejects the card	...the system cancels the transaction and ejects the card.	
5	Use present tense only.	The system <i>ejects</i> the card.	The system <i>ejected</i> the card.	
6	Use active voice rather than passive voice in flow of events	The system <i>ejects</i> the card.	The card <i>is ejected</i> .	
7	Clearly describe the interaction between the system and actors.	ATM customer enters PIN number <i>to the system</i> .	Customer enters PIN.	

Table 56 Restrictions (part 2)

#	Description	Example	
		RIGHT	WRONG
8	Use declarative sentence only.	The system ejects the card.	Ejects the card.
9	Use words in a consistent way.	ATM customer inserts the ATM card...	Customer inserts the ATM card...
10	Don't use modal verbs (e.g., <i>might</i> , <i>could</i> , <i>should</i>).	The system ejects the card.	The system <i>might</i> eject the card.
11	Avoid adverbs, such as <i>very</i> , <i>more</i> , <i>rather</i> , and <i>the like</i> .	The system ejects the card.	The system <i>very likely</i> ejects the card.
12	Use simple sentences only. A simple sentence must contain only one subject and one predicate. No compound subject and predicate are allowed. This restriction does not apply to the sentences using keywords: IF-THEN-ELSE-ELSEIF pairs, DO-UNTIL, MEANWHILE, and VALIDATES THAT.	1. The system displays ATM customer accounts. 2. The system prompts ATM customer for ...	System displays customer accounts <i>and</i> prompts customer for transaction type...
13	Don't use negatively adverb or adjective such as <i>hardly</i> and <i>never</i> . But it is allowed to use <i>not</i> or <i>no</i> .	The PIN number <i>has not</i> been validated	The PIN number <i>has never been</i> validated.
14	Don't use pronouns such as <i>he</i> , <i>his</i> , <i>him</i> , <i>himself</i> , <i>this</i> , <i>that</i> , <i>everyone</i> , <i>one</i> , <i>another</i> , <i>each</i> , <i>everything</i> , <i>many</i> , <i>it</i> , etc.	... <i>the system</i> reads the card number.	... <i>it</i> reads the card number.
15	Don't use participle phrases as adverbial modifier.	The system is idle. The system is displaying a Welcome message.	ATM is idle, displaying a Welcome message.
16	Use "the system" to refer to the system under design consistently.	the system	"ATM" or "The ATM system"

Table 57 Restrictions (part 3)

#	Description	Grammar	Explanation	Example	
				RIGHT	WRONG
17	Use keywords INCLUDE USE CASE to describe the include dependencies with other use cases.	INCLUDE USE CASE <included use case name>	The keywords can be used in basic and step alternative flows.	INCLUDE USE CASE Validate PIN	Include Validate PIN abstract use case.
18	Use keywords EXTENDED BY USE CASE to refer to the extended use case.	EXTENDED BY USE CASE <extending use case> or <specific use case> Appears either in the action part of a THEN or ELSE in the main flow, or as an action in an alternative flow.	The keywords can be used in basic and alternative flows.	EXTENDED BY USE CASE CreateIncident	Use case CreateIncident extends the current use case.
19	Use keyword RFS in a specific (or bounded) alternative flow to refer to a step number (or a lower bound step number and an upper bound step number) of a basic flow step that this alternative flow corresponds to.	RFS <basic flow step #> (specific alternative flow) RFS <set of step numbers> (specific alternative flow) Not required notation for global alternative flow.	One specific or bounded alternative flow must correspond to exactly one or more than one basic flow steps.	RFS 5 ... RFS 5-7, 10, 14 ...	
20	Use pairs of keywords of IF, THEN, ELSE, ELSEIF, and ENDIF to describe conditional logic sentences.	IF <condition> THEN <steps> ENDIF IF <condition> THEN <steps> - ELSE <steps> ENDIF IF <condition> THEN <steps> - ELSEIF <condition> THEN <steps> ENDIF	Appears in one flow only.	IF the system recognizes the ATM card, THEN the system reads the ATM card number.	If the system recognizes the card, it reads the card number.

Table 58 Restrictions (part 4)

#	Description	Grammar	Explanation	Example	
				RIGHT	WRONG
21	Use keyword MEANWHILE to describe concurrency.	<action> MEANWHILE <action>	It implies that the sentence before keyword MEANWHILE and the sentence after the keyword occur concurrently.	...the system cancels the transaction and MEANWHILE the system ejects the card.	...the system cancels the transaction and ejects the card.
22	Use keywords VALIDATES THAT to describe condition checking sentences. VALIDATES THAT means that the condition is evaluated and must be true to proceed to the next step.	VALIDATES THAT <condition>	The alternative case (the condition does not hold) must be described in its corresponding alternative flow (RFS).	...the system VALIDATES THAT the user entered PIN...	... the system checks whether the user-entered PIN...
23	Use keywords DO and UNTIL to describe iteration.	DO <steps> UNTIL <condition >	Following keyword DO is a sequence of steps. Following keyword UNTIL is a loop ending condition.	1. DO 2. action1 3. action2 4. UNTIL condition	
24	Use keyword ABORT to describe an exceptionally exit action. An alternative flow ends either with ABORT or RESUME STEP.	ABORT	Used in alternative flows, iterative, and conditional logic sentences. It means the ending of a use case.		
25	Use keywords RESUME STEP to describe the situation where an alternative flow goes back to its corresponding basic flow. An alternative flow ends either with ABORT or RESUME STEP.	RESUME STEP <basic flow step #>	Used in alternative flows.	RESUME STEP 5	
26	Each basic flow and alternative flow should have their own postconditions.	Refer to restriction # 19.			

Appendix B Dictionary of UCMeta

This appendix specifies the metaclasses of UCMeta (except the sentence patterns which are however specified and formalized in Appendix C), listed in alphabetical order. The description for each metaclass is broken down into sub-sections corresponding to different aspects. The following sub-sections and conventions are used to specify a metaclass.

- The *heading* gives the name of the metaclass.
- The “Description” sub-section gives a brief textual description of the meaning of a metaclass.
- All the direct generalizations of the metaclass are listed in the “Generalizations” subsection.
- A detailed description of the purpose of the metaclass is given in the “Purpose” subsection.
- The “Attributes” sub-section lists each of the attributes that are defined for that metaclass.
- Each attribute is specified by its name, its type, and multiplicity. This is followed by a textual description of the purpose and meaning of the attribute.
- The “Associations” sub-section lists all the association ends owned by the metaclass. The format for these is the same as the one for attributes described above.

B.1 AbortSentence

Description

`AbortSentence` is a type of `SpecialSentence`. It refers to special sentences containing keyword ABORT in UCSs.

Generalizations

- `SpecialSentence`

B.2 ActionSentence

Description

This metaclass represents one of the common properties of all action steps of a UCS. It is opposite to metaclass `Condition`.

Generalizations

- `Function`

B.3 ActionVerb

Description

As one type of verbs, an `ActionVerb` shows the performance of an action.

Generalizations

- `Verb`

B.4 ActorSubj

Description

An instance of the metaclass is the subject of a sentence, which refers to an actor of a use case.

Generalizations

- `Subject`

B.5 Adjective

Description

As one type of parts of speech, an `Adjective` is usually applied before a noun to modify or describe it, adding more information to make the meaning of the noun more clear and precise.

Generalizations

- `PartOfSpeech`

B.6 AdjectivePhrase

Description

An `AdjectivePhrase` is one type of `Phrases`, which is headed by an `Adjective`. Its pre-head-strings can only be an `AdverbPhrase`. Its post-head-strings can be an `InfinitivePhrase` or a `PrepositionalPhrase`. It functions as a pre-head-string of a `NounPhrase`, an object adjective complement, or a subject adjective complement.

Generalizations

- `Phrase`

Associations

- head: Adjective [1]
The head of the adjective phrase
- pre-Head-String: AdverbPhrase [0..1]
The pre-head-string of the adjective phrase
- post-Head-String: APPostHeadString [0..1]
The post-head-string of the adjective phrase

Examples

It is *too late to change the plan*. “late”, “too”, and “to change the plan” are the head, the pre-head-string, and the post-head-string of the adjective phrase, respectively.

B.7 Adverb

Description

Adverb is one of the parts of speech. It is used to describe a Verb, an Adjective, or another Adverb.

Generalizations

- PartOfSpeech

Examples

That lady dances *beautifully*.

B.8 AdverbPhrase

Description

An AdverbPhrase is one type of Phrases, which is headed by an Adverb. Its pre-head-strings can only be another AdverbPhrase. It does not have post-head-string. It functions as a pre-head-string of an AdjectivePhrase, an adverb modifier, or a pre-head-string of a VerbPhrase.

Generalizations

- Phrase

Associations

- head: Adverb [1] The head of the adverb phrase.
- pre-Head-String: Adverb [0..1] The pre-head-string of the adverb phrase.

Examples

She left the city *quite suddenly*. “quite” is the pre-head-string and “suddenly” is the head of the adverb phrase.

B.9 AdvModifier

Description

An `AdvModifier` modifies a sentence as an adverb phrase.

Generalizations

- `Modifier`

Associations

- `form: AdverbPhrase [1]`

The form of the adverb modifier

Examples

She left the city *quite suddenly*.

B.10 AlternativeFlow

Description

In a UCS, an `AlternativeFlow` is the field that gives the alternatives and exceptions to the reference flow (the basic flow or another alternative flow) it corresponding to. To compare with the reference flow, the alternative flow usually describes the less-common paths that need to address.

Associations

- `rfs: Sentence [*]`

The specific step of the reference flow, from which the alternative branches

- `condition: Sentence [0..1]`

The condition of the alternative flow

B.11 AND

Description

This metaclass refers to the conjunction: *AND*.

Generalizations

- `Conjunction`

B.12 APostHeadString

Description

`APostHeadString` is an abstract class. It is used to organize different types of post-head-strings of an adjective phrase.

B.13 Article

Description

An `Article` is combined with a `Noun` to indicate the type of reference being made by the noun. There are three articles in English language: *the*, *an*, and *a*.

Generalizations

- `Determiner`

B.14 BasicFlow

Description

In a UCS, its `BasicFlow` is the field that describes the most commonly executed path and the main functionality of the use case. It is also called “happy path” or “happy scenario”.

B.15 BoundedAlternative

Description

`BoundedAlternative` is a type of `AlternativeFlows`, where there is an alternative that occurs between two specified steps or more than one specified steps of the reference flow of the use case.

Generalizations

- `AlternativeFlow`

Constraints

- [1] A bounded alternative flow corresponds to more than one steps of its corresponding reference flow.

```
self.bfs->size()>1
```

B.16 BriefDescription

Description

In a UCS, the brief description is the field that provides the brief description of the use case.

Generalizations

None

Attributes

- `content: String`
The content of the brief description

Associations

- `sentences: Sentence[*]`

The sentences of the brief description

B.17 BY

Description

This metaclass refers to the preposition: *by*.

Generalizations

- Preposition

B.18 ColonList

Description

An instance of this metaclass uses the punctuation mark colon to introduce a list.

Associations

- `elements: ColonListElement [1..*]`
The elements of the colon list
- `conjunctions: Conjunction [*]`
The conjunctions connecting the elements of the colon list

Examples

The system prompts customer for transaction type: *Withdrawal, Query, or Transfer*.

B.19 ColonListElement

Description

An instance of this metaclass describe the elements, separated by conjunctions *and* or *or*, in a colon list.

Attributes

- `form: NounPhrase`
The form of the elements

B.20 Complement

Description

The abstract metaclass `Complement` represents the part of a sentence that gives more information about the `Subject` (as a subject complement) or the `Object` (as an object complement) of the sentence.

B.21 ComplexSentence

Description

A `ComplexSentence` refers to a sentence containing one independent clause and at least one dependent clause. In UCSs, the most commonly occurring complex sentences are `ConditionalSentences`, `IterativeSentences`, `ParallelSentences`, and `ConditionCheckSentences`, which are subtypes of the abstract metaclass `ComplexSentence`.

Generalizations

- `Sentence`

B.22 CompoundObject

Description

A compound object is composed of two or more objects, which are connected by one or more `Conjunctions`.

Generalizations

- `Object`

Associations

- `combinedObjects: Object [2..*] {ordered}`
The objects constituting the `CompoundObject`
- `conjunctions: Conjunction [1..*] {ordered}`
The conjunctions connecting the `combinedObjects`

Examples

Alex passed *all the projects and exams*.

B.23 CompoundPredicate

Description

A compound predicate is composed of two or more predicates, which are connected by one or more `Conjunctions`.

Generalizations

- `Predicate`

Associations

- `combinedPredicates: Predicate [2..*] {ordered}`
The predicates constituting the `CompoundPredicate`
- `conjunctions: Conjunction [1..*] {ordered}`
The conjunctions connecting the `combinedPredicate`

B.24 ConditionSentence

Description

This metaclass refers to one of the properties of condition sentences of a UCS including precondition and postcondition sentences. It is opposite to the metaclass `Action`.

Generalizations

- `Function`

B.25 ConditionalSentence

Description

As a type of complex sentences, a conditional sentence refers to a sentence containing keyword `IF-THEN-ELSE-ELSEIF-ENDIF`.

Generalizations

- `ComplexSentence`

Associations

- `IFcondition: Sentence [1]`
The IF condition of the complex sentence
- `THENactions: Sentence [1..*] {ordered}`
The THEN actions
- `ELSEIFcondition: Sentence [0..1]`
The ELSEIF condition
A conditional sentence may or may not have an ELSEIF condition.
- `ELSEactions: Sentence [*] {ordered}`
The ELSE actions

Examples

IF the employee is authenticated, THEN the system displays a Welcome message, ENDIF

B.26 ConditionCheckSentence

Description

As a type of complex sentences, a condition check sentence refers to a sentence including keyword `VALIDATES THATS`.

Generalizations

- `ComplexSentence`

Associations

- condition: Sentence [1]

The condition of the sentence

Examples

The system VALIDATES THAT the ATM card is expired.

B.27 Conjunction

Description

Conjunction is one of parts of speech. It connects two words, phrases, or clauses together. The most commonly used conjunctions are “and” and “or”. In UCMeta, conjunctions are used to construct CompoundPredicates and CompoundObjects.

Generalizations

- PartOfSpeech

Examples

I like apples *and* oranges.

B.28 CreateActionVerb

Description

As one type of action verbs, CreateActionVerb represents a set of verbs that are synonyms of verb “create”, such as “generate” and “establish”.

Generalizations

- ActionVerb

Examples

The system *creates* a new account for the customer.

B.29 DefiniteArticle

Description

A DefiniteArticle is one type of Articles and it refers to a particular member of a group.

Generalizations

- Article

Examples

The system informs *the* employee the success of the transaction.

B.30 Determiner

Description

A `Determiner` is a `Noun` modifier. It indicates the reference of a noun or a noun phrase. It is one of parts of speech and includes articles, numerals, etc.

Generalizations

- `PartOfSpeech`

B.31 DestroyActionVerb

Description

As one type of action verbs, `DestroyActionVerb` represents verbs that are synonymies of verb “destroy”, such as “kill” and “exterminate”.

Generalizations

- `ActionVerb`

B.32 DirectActorObj

Description

Such a direct object refers to an actor of a UCS.

Generalizations

- `DirectObject`

B.33 DirectObject

Description

`DirectObject` is one type of `Objects`. It answers the question of “Whom” or “What”.

Generalizations

- `Object`

Associations

- `objColonList: ColonList [0..1]`

The colon list following the direct object

Examples

Life always gives me *surprise*.

B.34 DirectSysObj

Description

Such a direct object refers to “the system” of a UCS.

Generalizations

- DirectObject

B.35 EdParticiplePhrase

Description

An EdParticiplePhrase is a type of ParticiplePhrases. Its head verb has a form of PastParticiple.

Generalizations

- ParticiplePhrase

Constraints

[1] The head of a participle phrase is a verb that has a form of past participle.

```
self.head.form = FormOfVerb::PastParticiple
```

Examples

Tom returned the *severely damaged* car.

B.36 ExistentialThere

Description

ExistentialThere denotes the “There” of a there-be sentence.

B.37 ExtendSentence

Description

ExtendSentence is a type of SpecialSentence. It represents the step (containing keyword EXTENDED BY USE CASE) in a UCS, from which the extending use case extends.

Generalizations

- SpecialSentence

Associations

- extendingUseCase: UML::UseCases::UseCase [1]

The extending use case of the sentence referring to

Examples

EXTENDED BY USE CASE Invalid PIN

B.38 FlowOfEvents

Description

This metaclass refers to the flows of events of a UCS. There are two types of flows of events in a UCS: the basic flow of event and one or more alternative flows.

Associations

- `steps`: `Sentence[1..*]{ordered}`
The ordered steps of the flow of events
- `postCondition`: `PostCondition[1]`
The postcondition of the flow of events

B.39 FROM

Description

This metaclass refers to the preposition: *from*.

Generalizations

- `Preposition`

B.40 FormOfVerb

Description

`FormOfVerb` is an enumeration type that specifies the literals for defining the form of a verb: `basic`, `passive`, `infinitive`, `past`, `present participle`, and `past participle`.

Description

- `Basic`
Indicates that the verb takes a basic form, e.g., “give”
- `Passive`
Indicates that the verb is a verb in a passive sentence, e.g., “is given”
- `Infinitive`
Indicates that the verb takes a basic form but in a infinitive phrase, e.g., “to give”
- `Past`
Indicates that the verb takes a past form, e.g., “gave”
- `PresentParticiple`
Indicates that the verb is the verb of a present participle phrase, e.g., “is giving”
- `PastParticiple`
Indicates that the verb is the verb of a past participle phrase, e.g., “has given”

B.41 Function

Description

This metaclass refers to one of the properties of sentences, which can either be action sentences or condition sentences.

B.42 GerundPhrase

Description

A GerundPhrase is a type of VerbPhrases. Its head Verb has a form of PresentParticiple. It can function as a Noun, acting as a Subject, Object, or PrepositionalObj.

Generalizations

- VerbPhrase

Associations

- logicSubj: NounPhrase[0..1]

The logic subject (a noun phrase) of the gerund phrase.

Constraints

- [1] The head of a gerund phrase is a verb that has a form of present participle.

```
self.head.form = FormOfVerb::PresentParticiple
```

Examples

After *brushing my teeth*, I went to bed.

B.43 GlobalAlternative

Description

GlobalAlternative is a type of alternative flows, where there is an alternative that can happen at any time and applies to all the steps of its reference flow.

Generalizations

- AlternativeFlow

Constraints

- [1] A global alternative flow corresponds to every step of its corresponding reference flow.

```
self.bfs->size()>0
```

B.44 IN

Description

This metaclass refers to the preposition: *in*.

Generalizations

- Preposition

B.45 IncludeSentence

Description

`IncludeSentence` is a type of `SpecialSentences`, containing keyword `INCLUDE USE CASE`. It is used in UCSs to describe an inclusion relationship between two use cases.

Generalizations

- `SpecialSentence`

Associations

- `includedUseCase: UseCase[1]`

The included use case of the sentence referring to

Examples

`INCLUDE USE CASE Validate PIN`

B.46 IndefiniteArticle

Description

An `InDefiniteArticle` is one type of articles and it refers to any member of a group. It can only be used before a singular `Noun`.

Generalizations

- `Article`

Examples

The system creates *a* new account for the customer.

B.47 IndirectActorObj

Description

`IndirectActorObj` is one type of `IndirectObjects`. This type of indirect objects refers to an actor.

Generalizations

- `IndirectObject`

B.48 IndirectObject

Description

`IndirectObject` is one type of `Objects`, which answers the question of “To whom”, “To what”, or “For what”.

Generalizations

- `Object`

Examples

Life always gives *me* surprise.

B.49 IndirectSysObj

Description

IndirectSysObj is one type of IndirectObjects, which refers to “the system” of a UCS.

Generalizations

- IndirectObject

B.50 InfinitivePhrase

Description

An InfinitivePhrase is a type of VerbPhrases. It contains the Infinitive form of the Verb, and can function as a Subject, an Object, and a post-head-string of a NounPhrase.

Generalizations

- VerbPhrase

Constraints

[1] The head of an infinitive phrase is a verb that has a form of ‘basic’.

```
self.head.form = FormOfVerb::Basic
```

Examples

To pass the exam is the only chance for Tom to pass the course.

B.51 IngParticiplePhrase

Description

An IngParticiplePhrase is a type of ParticiplePhrases. Its head verb has a form of PresentParticiple. It may contain a NounPhrase as its object.

Generalizations

- ParticiplePhrase

Associations

- logicSubj: NounPhrase [0..1]

The logic subject (a noun phrase) of the present participle phrase.

Constraints

[1] The head of a present participle phrase is a verb that has a form of present participle.

```
self.head.form = FormOfVerb::PresentParticiple
```

Examples

The student *sitting in that table* is one of my lab mates.

B.52 Initiation

Description

As one type of Transactions, an Initiation sentence refers to a sentence describing that the primary actor sends request and data to the system.

Generalizations

- Transaction

B.53 INORDERTO

Description

This metaclass refers to the special type of preposition: *in order to*.

Generalizations

- Preposition

Examples

The insect pretends to be dead *in order to* protect itself.

B.54 InternalTransaction

Description

As one type of Transactions, an InternalTransaction sentence refers to the sentence describing that the system alters its internal state (e.g., recording or modifying something).

Generalizations

- Transaction

B.55 IterativeSentence

Description

IterativeSentence is a type of ComplexSentences. It is composed of a condition and a set of actions, which are repeated when the condition is hold. In UCSs, sentences with keyword DO-UNTIL are iterative sentences.

Generalizations

- ComplexSentence

Associations

- WHILEcondition: Sentence [1]
The condition of the iterative sentence
- DOactions: Sentence [1..*] {ordered}

The sentences describing the actions repeated when the WHILEcondition is satisfied.

B.56 LinkingVerb

Description

As one type of Verbs, a LinkingVerb describes the state of being of the Subject of a sentence, followed by the SubjectComplement of the sentence. It does not express action. Instead, it describes a condition. The most common linking verb is *to be*. Other linking verbs include *appear, become, feel, look, seem, smell, sound, taste, and turn*.

Generalizations

- Verb

Examples

The system *is* idle.

B.57 Noun

Description

A Noun is one type of parts of speech, which can function as a Subject, an Object, a subject complement, an object complement of a Sentence, the object of a PrepositionPhrase, the head of a NounPhrase, ect.

Generalizations

- PartOfSpeech

Attributes

- number: NumberOfNoun

A noun can take two forms: Plural and Singular.

- basis: String

The basis form of a noun when the noun is plural

Examples

The system updates the customer's *account*.

B.58 NounFunctionForm

Description

NounFunctionForm is an abstract class. It is used to organize three types of Phrases (NounPhrase, GerundPhrase, and InfinitivePhrase) that can function as a Noun.

B.59 NounPhrase

Description

A NounPhrase is a type of Phrases. It has a Noun as its head. It may have pre-Head-Strings and post-Head-Strings. Its pre-head-strings can be a Determiner, an AdjectivePhrase, a ParticiplePhrase, a PossessiveNoun, and/or another Noun. Its post-head-strings can be a ParticiplePhrase, a PrepositionalPhrase, or an InfinitivePhrase. A noun phrase can function as a Subject, an Object, a PrepositionalObj, a Complement, and the object of a GerundPhrase, EdParticiplePhrase, or IngParticiplePhrase.

Generalizations

- Phrase
- NPPreHeadString
- NounFunctionForm
- PrepositionalObj

Attributes

- actorPosition: NPActorPositionType
If the noun phrase contains an actor, then this attribute indicates the position of the actor in the noun phrase.
- sysPosition: NPSysPositionType
If the noun phrase contains “the system”, then this attribute indicates the position of the string of “the system” in the noun phrase.

Associations

- containedActor: Actor[0..1]
If the noun phrase contains an actor, then the association refers to the actor.
- head: Noun[1]
The head of the noun phrase
- pre-Head-String: NP-Pre-Head-String[*]
The pre-head-string of the noun phrase
- post-Head-String: NP-Post-Head-String[*]
The post-head-string of the noun phrase

Examples

Tao's hometown is at the shore of beautiful Yellow River.

B.60 NPPostHeadString

Description

`NPPostHeadString` is an abstract class. It is used to organize different types of post-head-strings of a noun phrase.

B.61 NPPreHeadString

Description

`NPPreHeadString` is an abstract class. It is used to organize different types of pre-head-strings of a noun phrase.

B.62 NumberOfNoun

Description

`NumberOfNoun` is an enumeration type that specifies the literals for defining the number of a noun: plural or singular

Description

- `Singular` Indicates that the noun is a singular noun
- `Plural` Indicates that the noun is a plural noun

B.63 Numeral

Description

A `Numeral` is a word in natural language to represents a number. Different numerals can represent the same number. For example, “9” and “nine” are different numerals.

Generalizations

- `Determiner`

B.64 ObjAdjPComplt

Description

`ObjAdjPComplt` is a type of `ObjectComplt`s. It takes an `AdjectivePhrase` as its form.

Generalizations

- `ObjectComplt`

Associations

- `form: AdjectivePhrase [1]`
The form of the object complement

Examples

He painted the wall *green*.

B.65 Object

Description

The Object of a Predicate denotes something that receives the action triggered by the Subject. It usually comes after the Predicator. Objects fall into two main categories: DirectObject and IndirectObject.

Associations

- form: NounFunctionForm[1]

The form of the object

Examples

He painted *the wall* green.

B.66 ObjectComplt

Description

The abstract class represents the part of a sentence that gives more information about the Object.

Generalizations

- Complement

Examples

They painted the house *red*.

B.67 ObjEdParticipleComplt

Description

This object complement takes a past participle as its form.

Generalizations

- ObjectComplt

Associations

- form: EdParticiplePhrase[1]

The form of the object complement

Examples

He paints the house *owned by his parents*.

B.68 ObjInfinitiveComplt

Description

This object complement takes an infinitive phrase as its form.

Generalizations

- ObjectComplt

Associations

- form: InfinitivePhrase[1]
The form of the object complement

Examples

He has the key *to open the door*.

B.69 ObjIngParticipleComplt

Description

This object complement takes a present participle as its form.

Generalizations

- ObjectComplt

Associations

- form: IngParticiplePhrase[1]
The form of the object complement

Examples

The system sends a message *displaying the login error*.

B.70 ObjPPComplt

Description

This object complement takes a present participle as its form.

Generalizations

- ObjectComplt

Associations

- form: PrepositionalPhrase[1]
The form of the object complement

Examples

He took the key *on the table*.

B.71 OR

Description

This metaclass refers to the conjunction: *or*.

Generalizations

- Conjunction

B.72 ParallelSentence

Description

`ParallelSentence` is a type of `ComplexSentences`. It is used to model concurrent sentences. It refers to sentences using keyword `MEANWHILE` in UCSs.

Generalizations

- `ComplexSentence`

Associations

- `parallelActions: SimpleSentence [2..*]`
The actions occurring concurrently

B.73 ParticiplePhrase

Description

A `ParticiplePhrase` is a type of `VerbPhrases`. Its head `Verb` has a form of `PresentParticiple` or `PastParticiple`. In terms of function, a participle phrase always functions as an adjective to modify or describe a `Noun` or as a `SubjectComplt`.

Generalizations

- `VerbPhrase`

B.74 PartOfSpeech

Description

`PartOfSpeech` is an abstract class, used to organize `Determiner`, `Noun`, `Verb`, `Adjective`, `Conjunction`, `Adverb`, and `Preposition`.

B.75 Phrase

Description

A `Phrase` is a group of related words that function as a unit in a sentence. Unlike clauses, a phrase does not have `Subject/Verb` combination. A phrase is composed of three parts: the head, pre-head-strings, and post-head-strings. The head of a phrase determines the function of the whole phrase in a sentence. The none-head part of the phrase is its pre-head-string if it appears before the head; otherwise the none-head part is the post-head-string of the phrase.

B.76 PhrasalVerb

Description

A `PhrasalVerb` is a verb plus a preposition or adverb to create a new meaning which is different with the original verb.

Generalizations

- `ActionVerb`

Examples

She *hanged up* the phone.

B.77 PossessiveNoun

Description

A `PossessiveNoun` is a `Noun` that changes its form to show its ownership relation with some other things.

Generalizations

- `Noun`

Examples

The *student's* name is Mike.

B.78 PossessiveVerb

Description

As one type of `ActionVerbs`, `PossessiveVerb` indicates possessions, ownership, etc. In UCSs, we commonly use possessive verbs: “have/has”, “(be) composed (of)”, “contain” and “consist (of)”.

Generalizations

- `ActionVerb`

Examples

The system *has* the information of the customer.

B.79 Postcondition

Description

For each flow of events of a use case, its `Postcondition` is a constraint that has to be true when the flow of events is finished.

Attributes

- `content: String`

It refers to all the sentences of the postcondition as a whole.

Associations

- `postConditionSenetences: Sentence [*]`

The sentences describing the postcondition

B.80 Precondition

Description

For a `UseCase`, its `Precondition` is a constraint that has to be true when the use case is invoked.

Attributes

- content: String

It refers to all the sentences of the precondition as a whole.

Associations

- preConditionSentences: Sentence [*]

The sentences describing the precondition

B.81 Predicate

Description

The predicate of a sentence is one of the two main parts of a sentence (the other is the subject). It modifies the subject of the sentence and tells what the subject has, does, or is. It must contain a `Predicator`, which takes a `VerbPhrase` as the form. It may or may not contain `Objects` and `Complements`.

Associations

- predicator: `Predicator` [1]

The verb of the predicate

- object: `Object` [0, 1, 2]

The object(s) of the sentence

- complement: `Complement` [*]

The complement(s) of the sentence

Examples

Life is tough.

B.82 Predicator

Description

The `Predicator` of a `Predicate` is the verb of the predicate.

Generalizations

None

Attributes

- isNegative: Boolean

This attribute indicates whether the sentence is negative or not.

Associations

- form: `Verb` [1]

The form of the predicator

Examples

Life *is* tough.

B.83 Preposition

Description

Preposition is one of parts of speech, which introduces PrepositionalPhrase. It does not modify words, but it shows important relationships of words. In English, the most commonly used prepositions are “of”, “in”, “to”, “for”, and “on”.

Generalizations

- PartOfSpeech

Examples

The dog is sleeping *under* the table.

B.84 PrepositionalObj

Description

PrepositionalObj is an abstract class, used to organize different types of objects of prepositional phrases.

Examples

Mike bought a gift to *his girlfriend*.

B.85 PrepositionalPhrase

Description

A PrepositionalPhrase is one type of Phrases, which is headed by a Preposition. It does not have a pre-head-string. Its post-head-string is its object, which can either be a NounPhrase or a GerundPhrase.

Generalizations

- Phrase
- NPPostHeadString

Associations

- head: Preposition [1]
The head of the prepositional phrase
- post-Head-String: PrepositionalObj [1]
The post-head-string of the prepositional phrase

Examples

Mike bought a gift *to his girlfriend*.

B.86 Response

Description

As one type of `Transaction`, a `Response` action refers to sentences describing that the system replies to the actor (either the primary actor or a secondary actor) with the result.

Generalizations

- `Transaction`

B.87 Response2PrimaryActor

Description

As one type of `Response` actions, a `Response2PrimaryActor` action refers to a sentence describing that the system replies to the primary actor with the result.

Generalizations

- `Response`

B.88 Response2SecondaryActor

Description

As one type of `Response` actions, a `Response2SecondaryActor` action refers to a sentence describing that the system replies to a secondary actor of the use case with the result.

Generalizations

- `Response`

B.89 ResumeStepSentence

Description

`ResumeStepSentence` is a type of `SpecialSentences`. It refers to sentences containing keyword `RESUME STEP BACK`.

Generalizations

- `SpecialSentence`

Associations

- `resumePoint: Sentence[1]`

The aggregation association refers to a specific step of the reference flow that the resume step sentence refers to.

Examples

RESUEM BACK TO #1

B.90 Sentence

Description

A `Sentence` is a grammatical unit of one or more words, which usually begins with a capital letter and ends with a full stop (period), a question mark, or an exclamation mark. This abstract class is the superclass of metaclasses: `SimpleSentence` and `ComplexSentence`.

Attributes

- `Content: String`
The content of the sentence

Associations

- `transactionType: Transaction[0..1]`
The transaction type of the sentence
- `functionType: Function[0..1]`
The function type of the sentence

B.91 SimpleSentence

Description

A `SimpleSentence` refers to a sentence containing one independent clause: one `Subject` and one `Predicate`.

Generalizations

- `Sentence`

Associations

- `subject: Subject [1]`
The subject of the simple sentence
- `predicate: Predicate [1]`
The predicate of the simple sentence
- `modifier: Modifier [0..1]`
The modifier of the simple sentence
- `pattern: SentencePattern [1]`
The pattern of the simple sentence

Examples

The sentence is a simple sentence.

B.92 Subject

Description

The Subject of a Sentence has the grammatical function to relate itself to any other parts of the sentence: Objects and Complements, through a Verb.

Associations

- form: NounFunctionForm [1]

The form of the subject

Examples

The subject of a sentence tells what the sentence is about.

B.93 SpecificAlternative

Description

SpecificAlternative is a type of AlternativeFlows, where there is an alternative that applies to one specific step of its reference flow.

Generalizations

- AlternativeFlow

Constraints

[1] A specific alternative flow corresponds to only one step of its corresponding reference flow.

```
self.bfs->size ()=1
```

B.94 SubjAdjComplt

Description

SubjAdjComplt is a type of SubjectComplts. It takes an adjective phrase as its form.

Generalizations

- SubjectComplt

Associations

- form: AdjectivePhrase [1]

The core of the subject complement

- postInf: InfinitivePhrase [0..1]

The infinitive phrase following the adjective phrase of the subject complement

- postPP: PrepositionalPhrase [0..1]

The prepositional phrase following the adjective phrase of the subject complement

Examples

She is *beautiful*.

B.95 SubjectComplt

Description

SubjectComplt is an abstract class, used to represents the part of the sentence that gives more information about the Subject. A subject complement follows a LinkingVerb; it is normally an AdjectivePhrase or a NounPhrase.

Generalizations

- Complement

B.96 SubjNPComplt

Description

SubjNPComplt is a type of SubjectComplts. It takes a NounPhrase as its form.

Generalizations

- SubjectComplt

Associations

- form: NounPhrase

The form of the subject complement

Examples

She is *an angel*.

B.97 SysSubj

Description

The SysSubj is one type of Subjects. It refers to “the system” of a use case when “the system” functions as the subject of a sentence.

Generalizations

- Subject

Examples

The system sends a message to the customer.

B.98 ThereBeSentence

Description

ThereBeSentence is a type of SimpleSentences. It is composed of ExistentialThere, a LinkingVerb (e.g., “Be”), and the true subject (ThereBePostSubject).

Generalizations

- SimpleSentence

Associations

- there: ExistentialThere [1]
The “There” of the sentence
- be: LinkingVerb [1]
The linking verb of the sentence
- subject: NounPhrase [1]
The true subject of the sentence

Examples

There is a red car.

B.99 TO

Description

This metaclass refers to the preposition: *to*.

Generalizations

- Preposition

Examples

He is going *to* graduate soon.

B.100 Transaction

Description

This abstract metaclass is used to organize different types of actions: Initiation, Validation, InternalTransaction, and Response. This classification is adapted from [21], except the fifth.

- Initiation: the primary actor sends request and data to the system to initiate the use case.
- Validation: the system validates the request and the data.
- InternalTransaction: the system alters its internal state (e.g., recording or modifying something).
- Response2PrimaryActor: the system replies to the actor with the result.
- Response2SecondaryActor: the system sends requests to a secondary actor.

B.101 UseCase

Description

In UML [77], `UseCase` is defined as “the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system.” A use case should describe the interactions between the system and the entities external to the system.

Generalizations

- `BehavioeredClassifier` (from [77])

Associations

- `specification: UseCaseSpecification[1]`
The specification of the use case

B.102 UCModel

Description

`UCModel` is a metaclass, used to model a use case model, which is composed of a set of use case model elements.

Generalizations

- `UseCaseModelElement`

Associations

- `modelElements: UseCaseModelElement [1..*]`
The model elements of the use case model

B.103 UseCaseModelElement

Description

Like metaclass `Element` of UML 2 [77], `UseCaseModelElement` is a constituent of the `UseCaseModel`. As such, it has the capability of owning other model elements. It is an abstract class with no superclass, and used as the common superclass for all metaclasses in our use case metamodel.

Attributes

- `name: String`
The name of the use case model element
- `description: String`
The description of the use case model element

B.104 UseCaseSpecification

Description

Each use case has a UCS. The metaclass refers to the textual description of the UCS.

Attributes

- `content: String`
The description of the UCS

Associations

- `briefDescription: BriefDescription[1]`
The brief description of the UCS
- `preCondition: PreCondition[1]`
The precondition of the UCS
- `flows: FlowOfEvents[1..*]`
The flows of events of the UCS
- `primaryActor: Actor[1]`
The actor that initiates the use case
- `secondaryActors: Actor[*]`
The actors relied on to accomplish the use case

B.105 Validation

Description

As one type of `Transactions`, a `Validation` action refers to one of the properties of the sentences describing the system validates the request and the data.

Generalizations

- `Transaction`

B.106 Verb

Description

A `Verb` is one type of parts of speech, which usually denotes an action, an event, or a state of being. There are two main kinds of verbs: `ActionVerbs` and `LinkingVerbs`.

Generalizations

- `PartOfSpeech`

Attributes

- `basis: String`
The basis of a verb

Associations

- `form: FormOfVerb[1]`

The form of the verb

B.107 VerbPhrase

Description

As one of `Phrases`, this metaclass has a `Verb` as its head and may have `pre-Head-Strings` and `post-Head-Strings`. Its `pre-head-strings` can only be an `AdverbPhrase`, and its `post-head-strings` can be a `PrepositionalPhrase` or a `NounPhrase` (only its three subtypes can have a noun phrase as their objects). If the head of the verb phrase has a basic verb form, then this type of verb phrases can function as `Predicators` of sentences.

Generalizations

- `Phrase`

Associations

- `head: Verb [1]`
The head of the verb phrase
- `pre-Head-String: AdverbPhrase [0..1]`
The pre-head-string of the verb phrase
- `post-Head-String: PrepositionalPhrase [0..1]`
The post-head-string of the verb phrase
- `verbalPhraseObj: NounPhrase [0..1]`
The object of the verb phrase

Examples

Mike *carefully peeled* the stamp off the envelope.

B.108 WITH

Description

This metaclass refers to the preposition: *with*.

Generalizations

- `Preposition`

Appendix C Specification/Formalization of Sentence Patterns

This appendix specifies classifications of each category of sentence patterns: *sv*, *svdo*, *sviodo*, *svdoc*, *slvsubjectcomplt*, *svc*, *svcc*, and *sviodoc*. It also specifies and formalizes each sentence pattern.

C.1 SV

Description:

A *sv* sentence is composed of only one Subject and one Predicator formed by an ActionVerb.

OCL expression:

```
Context SimpleSentence
self.sentencePattern.OclIsTypeOf(SV) implies
    self.subject->size()=1
    and
    self.predicate.predicator.form.head.OclIsTypeOf(ActionVerb)
    and self.predicate.object.size()=0
    and self.predicate.complement.size()=0
```

Taxonomy:

The taxonomy of this sentence pattern is described in Figure 63. If the subject of a sentence is 'the system', then the sentence pattern of the sentence is an instance of the *SysSubjV*. If the subject of the sentence is an actor of the use case model, then the sentence pattern of the sentence is an instance of the *ActorSubjV*.

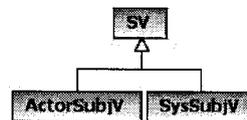


Figure 63 Taxonomy of sentence pattern *sv*

C.1.1.1 SysSubjV

Description:

The subject of a *sv* sentence is "the system" of the UCS containing this sentence.

OCL expression:

```
Context SimpleSentence
self.sentencePattern.OclIsTypeOf(SysSubjV) implies
    self.subject.OclIsTypeOf(SysSubj)
```

Example:

The system exists.

C.1.1.2 ActorSubjV**Description:**

The subject of a sv sentence is one of the actors of the UCS containing the sentence.

OCL expression:

```

Context SimpleSentence
    self.sentencePattern.OclIsTypeOf (ActorSubjV) implies
        self.subject.OclIsTypeOf (ActorSubj)

```

Example:

The customer logged in.

C.2 SVDO**Description:**

A SVDO sentence contains one Subject, one Predicator (an ActionVerb), and one DirectObject.

OCL expression:

```

Context SimpleSentence
    self.sentencePattern.OclIsTypeOf (SVDO) implies
        self.subject->size ()=1
        and
        self.predicate.predicator.form.head.OclIsTypeOf (ActionVerb)
        and self.predicate.object->size ()=1
        and self.predicaae.object->exists (OclAsType (DirectObject))
        and self.predicate.complement->size ()=0

```

Taxonomy:

The taxonomy of this sentence pattern is described in Figure 64. If the subject of a sentence is 'the system', then the sentence pattern of the sentence is an instance of the SysSubjVDO. If the subject of the sentence is an actor of the use case model, then the sentence pattern of the sentence is an instance of the ActorSubjVDO. If the direct object of the sentence, whose sentence pattern is SysSubjVDO, is an actor, then the sentence has an instance of SysSubjVDirectActorObj as its sentence pattern, more specifically.

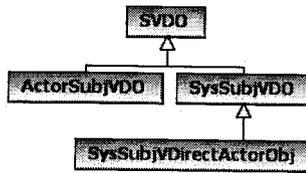


Figure 64 Taxonomy of sentence pattern svdo

C.2.1.1 SysSubjVDO

Description:

The subject of a svdo sentence is “the system” of the UCS containing this sentence.

OCL expression:

```

Context SimpleSentence
self.sentencePattern.OclIsTypeOf(SysSubjVDO) implies
self.subject.OclIsTypeOf(SysSubj)
  
```

Example:

The system recognizes the ATM card.

C.2.1.2 SysSubjVDirectActorObj

Description:

The direct object of a SysSubjVDO sentence is one of the actors of the UCS containing this sentence.

OCL expression:

```

Context SimpleSentence
self.sentencePattern.OclIsTypeOf(SysSubjVDirectActorObj)
implies
self.objects.OclIsTypeOf(DirectActorObj)
  
```

Example:

The system informs Employee.

C.2.1.3 ActorSubjVDO

Description:

The subject of a svdo sentence is one of the actors of the UCS containing this sentence.

OCL expression:

```

Context SimpleSentence
self.sentencePattern.OclIsTypeOf(ActorSubjVDO) implies
self.subject.OclIsTypeOf(ActorSubj)
  
```

Example:

Customer informs the system.

C.3 SVIDO

Description:

A SVIDO sentence contains one subject, one action verb, one direct object, and one indirect object.

OCL expression:

```
Context SimpleSentence
self.sentencePattern.OclIsTypeOf(SVIDO) implies
self.subject->size()=1
and
self.predicate.predicator.form.head.OclIsTypeOf(ActionVerb)
and self.predicate.object->size()=2
and self.predicate.object->exists(OclAsType(DirectObject))
and self.predicate.object->exists(OclAsType(IndirectObject))
and self.predicate.complement->size()=0
```

Taxonomy:

The taxonomy of this sentence pattern is described in Figure 65. If the indirect object of a sentence is ‘the system’, then the sentence pattern of the sentence is an instance of the SVSysIODO. If the indirect object of the sentence is an actor of the use case model, then the sentence pattern of the sentence is an instance of the SVActorIODO. According to the different types of the sentence subject, the sentence pattern SVSysIODO is further classified into SysSubjVSysIODO and ActorSubjVSysIODO, and the sentence pattern SVActorIODO is further classified into SysSubjVActorIODO and ActorSubjVActorIODO.

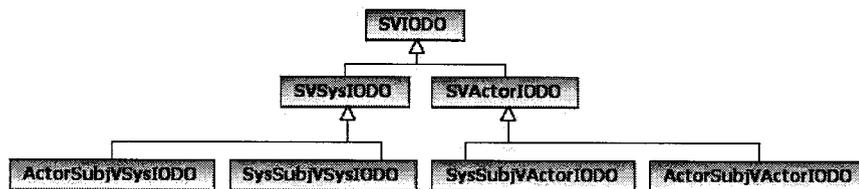


Figure 65 Taxonomy of sentence pattern SVIDO

C.3.1.1SVSysIODO

Description:

The indirect object of a SVSysIODO sentence is “the system” of the UCS containing the sentence.

OCL expression:

```
Context SimpleSentence
self.sentencePattern.OclIsTypeOf(SVSysIODO) implies
```

```
self.predicate.objects->select (OclIsTypeOf (IndirectObject)) .  
    OclIsTypeOf (IndirectSysObj)
```

Example:

The employee informs the system the completion of the action.

C.3.1.2 SysSubjV SysIODO

Description:

The subject of a svSysIODO sentence is “the system” of the UCS containing the sentence.

OCL expression:

```
Context SimpleSentence  
self.sentencePattern.OclIsTypeOf (SysSubjV SysIODO) implies  
self.subject.OclIsTypeOf (SysSubj)
```

Example:

The system informs the system the completion of the transaction.

C.3.1.3 ActorSubjV SysIODO

Description:

The subject of a svSysIODO sentence is one of the actors of the UCS containing the sentence.

OCL expression:

```
Context SimpleSentence  
self.sentencePattern.OclIsTypeOf (ActorSubjV SysIODO)  
implies self.subject.OclIsTypeOf (ActorSubj)
```

Example:

The customer informs the system the completion of the input.

C.3.1.4 SVActorIODO

Description:

The indirect object of a sviODO sentence is one of the actors of the UCS containing the sentence.

OCL expression:

```
Context SimpleSentence  
self.sentencePattern.OclIsTypeOf (SVActorIODO) implies  
self.predicate.objects->select (OclIsTypeOf (IndirectObject)) .  
    OclIsTypeOf (IndirectActorObj)
```

C.3.1.5 SysSubjV ActorIODO

Description:

The subject of a SVActorIODO sentence is “the system” of the UCS containing the sentence.

OCL expression:

```
Context SimpleSentence
    self.sentencePattern.OclIsTypeOf (SysSubjVActorIODO) implies
        self.subject.OclIsTypeOf (SysSubj)
```

Example:

The system informs the customer the completion of the transaction.

C.3.1.6 ActorSubjVActorIODO**Description:**

The subject of a SVActorIODO sentence is one of the actors of the UCS containing the sentence.

OCL expression:

```
Context SimpleSentence
    self.sentencePattern.OclIsTypeOf (ActorSubjVActorIODO)
    implies self.subject.OclIsTypeOf (ActorSubj)
```

Example:

The employee informs the customer the completion of the transaction.

C.4 SVD OC**Description:**

A SVD OC sentence contains one subject, one action verb, one direct object, and one object complement.

OCL expression:

```
Context SimpleSentence
    self.sentencePattern.OclIsTypeOf (SVD OC) implies
        self.subject->size ()=1
        and self.predicate.predicator.form.head.OclIsTypeOf (ActionVerb)
        and self.predicate.objects->size ()=1
        and self.predicate.objects->exists (OclAsType (DirectObject))
        and self.predicate.complement->size ()=1
        and self.predicate.complement.OclIsTypeOf (ObjectComplt)
```

Taxonomy:

The taxonomy of this sentence pattern is described in Figure 66. According to the different types of object complements, this sentence pattern is classified into the categories: SVDOObjAdjComplt, SVDOObjPPComplt, SVDOObjEdParticipleComplt, SVDOObjInfinitiveComplt, and SVDOObjIngParticipleComplt, which are further classified into subtypes, according to the type of the sentence subject: either the system or an actor.

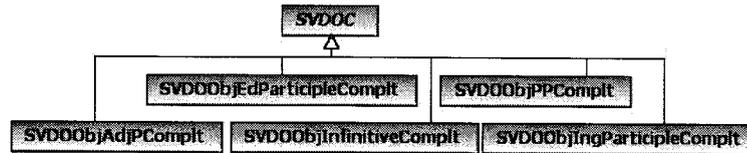


Figure 66 Taxonomy of sentence pattern svdoc

C.4.1.1SVDOObjAdjComplt

Description:

The object complement of a SVDOC sentence is an AdjectivePhrase.

OCL expression:

```

Context SimpleSentence
self.sentencePattern.OclIsTypeOf (SVDOObjAdjComplt) implies
self.predicate.complement.OclIsTypeOf (ObjAdjComplt)
  
```

C.4.1.2SVDOObjObjEdParticipleComplt

Description:

The object complement of a SVDOC sentence is an EdParticiplePhrase.

OCL expression:

```

Context SimpleSentence
self.sentencePattern.OclIsTypeOf (SVDOObjEdParticipleComplt)
implies
self.predicate.complement.OclIsTypeOf (ObjEdParticipleComplt)
  
```

Example:

The system saves the data inputted by the employee.

C.4.1.3SVDOObjIngParticipleComplt

Description:

The object complement of a SVDOC sentence is an IngParticiplePhrase.

OCL expression:

```

Context SimpleSentence
self.sentencePattern.OclIsTypeOf (SVDOObjIngParticipleComplt)
implies
  
```

```
self.predicate.complement.OclIsTypeOf (ObjIngParticipleComplt)
```

Example:

The system records the data containing the customer's information.

C.4.1.4SVDOObjPPComplt

Description:

The object complement of a SVDOC sentence is a PropositionalPhrase.

OCL expression:

```
Context SimpleSentence
self.sentencePattern.OclIsTypeOf (SVDOObjPPComplt) implies
self.predicate.complement.OclIsTypeOf (ObjPPComplt)
```

Example:

The system displays a picture in black and white.

C.4.1.5SVDOObjInfinitiveComplt

Description:

The object complement of a SVDOC sentence is an InfinitivePhrase.

OCL expression:

```
Context SimpleSentence
self.sentencePattern.OclIsTypeOf (SVDOObjInfinitiveComplt)
implies
self.predicate.complement.OclIsTypeOf (ObjInfinitiveComplt)
```

Example:

The system displays a message to explain the transaction details.

C.5 SLVSubjectComplt

Description:

This type of sentences contains one subject, one linking verb, and one subject complement.

OCL expression:

```
Context SimpleSentence
self.sentencePattern.OclIsTypeOf (SLVSubjectComplt) implies
self.subject->size()=1
and
self.predicate.predicator.form.head.OclIsTypeOf (LinkingVerb)
and self.predicate.objects->size()=0
and self.predicate.complement->size()=1
```

```
and self.predicate.complement.OclIsTypeOf (SubjectComplt)
```

Taxonomy:

The taxonomy of this sentence pattern is described in Figure 67. If the subject complement of a sentence is an instance of `SubjNPComplt`, then the sentence pattern of the sentence is an instance of `SLVSubjNPComplt`. If the subject complement of the sentence is an instance of `SubjAdjPComplt`, then the sentence pattern of the sentence is an instance of `SLVSubjAdjPComplt`. These two subtypes are further classified according to the type of the sentence subject: either the system or an actor.

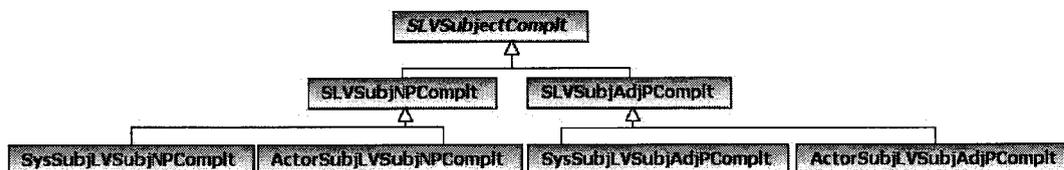


Figure 67 Taxonomy of sentence pattern `SLVSubjectComplt`

C.5.1.1 SLVSubjNPComplt

Description:

The subject complement of a `SLVSubjectComplt` sentence is a `NounPhrase`.

OCL expression:

```
Context SimpleSentence
self.sentencePattern.OclIsTypeOf (SLVSubjNPComplt) implies
self.predicate.complement.OclIsTypeOf (SubjNPComplt)
```

Examples:

The customer is an employee.

C.5.1.2 SysSubjLVSubjNPComplt

Description:

The subject complement of a `SLVSubjNPComplt` sentence is “the system” of the UCS containing the sentence.

OCL expression:

```
Context SimpleSentence
self.sentencePattern.OclIsTypeOf (SysSubjLVSubjNPComplt) implies
self.subject.OclIsTypeOf (SysSubj)
```

Examples:

The system is the data processing center of the bank.

C.5.1.3 ActorSubjLVSubjNPComplt

Description:

The subject complement of a SLVSubjNPComplt sentence is one of the actors of the UCS containing the sentence.

OCL expression:

```
Context SimpleSentence
self.sentencePattern.OclIsTypeOf (ActorSubjLVSubjNPComplt)
implies
self.subject.OclIsTypeOf (ActorSubj)
```

Examples:

The employee is a senior employee.

C.5.1.4SLVSubjAdjComplt

Description:

The subject complement of a SLVSubjectComplt sentence is an AdjectivePhrase.

OCL expression:

```
Context SimpleSentence
self.OclIsTypeOf (SLVSubjAdjComplt) implies
self.predicate.complement.OclIsTypeOf (SubjAdjComplt)
```

Examples:

The system is idle.

C.5.1.5SysSubjLVSubjAdjComplt

Description:

The subject complement of a SLVSubjAdjComplt sentence is “the system” of the UCS containing the sentence.

OCL expression:

```
Context SimpleSentence
self.sentencePattern.OclIsTypeOf (SysSubjLVSubjAdjComplt)
implies
self.subject.OclIsTypeOf (SysSubj)
```

Examples:

The system is idle.

C.5.1.6ActorSubjLVSubjAdjComplt

Description:

The subject complement of a SLVSubjAdjComplt sentence is one of the actors of the UCS containing the sentence.

OCL expression:

Context SimpleSentence

```
self.sentencePattern.OclIsTypeOf (ActorSubjLVSubjAdjComplt)
implies
self.subject.OclIsTypeOf (ActorSubj)
```

Examples:

The employee is busy.

C.6 SVC

Description:

The sentence contains one subject, one action verb, and one object complement.

OCL expression:

Context SimpleSentence

```
self.sentencePattern.OclIsTypeOf (SVC) implies
self.subject->size()=1
and self.predicate.predicator.form.head.OclIsTypeOf (ActionVerb)
and self.predicate.objects->size()=0
and self.predicate.complement->size()=1
and self.predicate.complement.OclIsTypeOf (ObjectComplt)
```

Examples:

The system wants to destroy the object.

C.7 SVCC

Description:

This type of sentences contains one subject, one action verb, and two object complements.

OCL expression:

Context SimpleSentence

```
self.sentencePattern.OclIsTypeOf (SVCC) implies
self.subject->size()=1
and self.predicate.predicator.form.head.OclIsTypeOf (ActionVerb)
and self.predicate.objects->size()=0
and self.predicate.complement->size()=2
```

Examples:

The system wants to destroy the object in order to release some memory.

C.8 SVIODOC

Description:

This type of sentences contains one subject, one action verb, one indirect object, one direct object and one object complement.

OCL expression:

Context SimpleSentence

```
self.sentencePattern.OclIsTypeOf(SVIODOC) implies  
self.subject->size()=1  
and self.predicate.predicator.form.head.OclIsTypeOf(ActionVerb)  
and self.predicate.objects->size()=2  
and self.predicae.object->exists(OclAsType(DirectObject))  
and self.predicate.object->exists(OclAsType(IndirectObject))  
and self.predicate.complement->size()=1  
and self.predicate.complement.OclIsTypeOf(ObjectComplt)
```

Examples:

The system informs the employee the completion of the transaction within two seconds.

Appendix D Specification/Formalization of Sentence Semantic Types

Whether a sentence is a condition or an action is identified by following the OCL expressions given in Figure 68. Five transaction types (Initiation, Validation, InternalTransaction, Response2PrimaryActor, and Response2SecondaryActor) can be identified by applying the rules specified as OCL expressions in Figure 69, Figure 70, Figure 71, Figure 72, and Figure 73, respectively.

```
Context Sentence
self.functionType.OclIsTypeOf(Condition) implies
self.preCondition->size() != 0
or self.flowOfEvents.postCondition->size() != 0
or self.OclIsTypeOf(ConditionalSentence)
or self.OclIsTypeOf(ConditionCheckSentence)
or self.OclIsTypeOf(IterativeSentence)
or self.flowOfEvents.OclAsTypeOf(AlternativeFlow).condition->size() != 0

self.functionType.OclIsTypeOf(Action) implies
self.OclAsTypeOf(SimpleSentence).flowOfEvents->size() != 0
or (self.OclIsTypeOf(ComplexSentence)
    and self.briefDescription->size() = 0)
or (self.OclIsTypeOf(ComplexSentence)
    and
    (self.OclAsTypeOf(conditionalSentence).eLSEactions->includes(self)
    or self.OclAsTypeOf(IterativeSentence).DOactions->includes(self)
    or self.OclAsTypeOf(ParallelSentence).parallelActions
    ->includes(self)))
```

Figure 68 Formal specification/identification of Function

```
Context Sentence
sentence.functionType.OclIsTypeOf(ActionSentence) and
self.transactionType.OclIsTypeOf(Initiation) implies
    self.OclAsTypeOf(SimpleSentence).subject.OclIsTypeOf(ActorSubj) and
    ((sentence.predicate.objects->size() > 0 and sentence.predicate.objects->
        exists(obj | obj.OclIsTypeOf(DirectSysObj))
    or
    (sentence.predicate.complements->size() > 0 and
    sentence.predicate.complements->
        exists(complt | complt.OclIsTypeOf(ObjPPComplt)
        and complt.content.isContainingSystem()))
    )
```

isContainingSystem() is used to check whether the content of the complement of the sentence contains the string "the system".

Figure 69 Formal specification/identification of Initiation

```

Context Sentence
sentence.functionType.OclIsTypeOf(ActionSentence) and
self.transactionType.OclIsTypeOf(Validation) implies
  self.sentencePattern.OclIsTypeOf(ConditionCheckSenence)

```

Figure 70 Formal specification/identification of Validation

```

Context Sentence
sentence.functionType.OclIsTypeOf(ActionSentence) and
self.transactionType.OclIsTypeOf(InternalTransaction) implies
  self.OclAsTypeOf(SimpleSentence).subject.OclIsTypeOf(SysSubj)
and
  (sentence.predicate.objects->size()==0
   or
   (sentence.predicate.objects->size())>0 and sentence.predicate.objects->
     exists(obj | not(obj.OclIsTypeOf(DirectActorObj))
           and not(obj.OclIsTypeOf(IndirectActorObj)))
   and
   (sentence.predicate.complements->size()==0
    or
    (sentence.predicator.complements->size())>0
     and sentence.predicate.complements->
       exists(complt | complt.OclIsTypeOf(ObjPPComplt)
             and complt.toActor->size()==0)
   )
  )
)
)
)

```

Figure 71 Formal specification/identification of InternalTransaction

Context Sentence

```
sentence.functionType.OclIsTypeOf(ActionSentence) and
self.transactionType.OclIsTypeOf(Response2PrimaryActor) implies
self.OclAsTypeOf(SimpleSentence).subject.OclIsTypeOf(SysSubj)
and
(
  (sentence.predicate.objects->size() != 0 and sentence.predicate.objects->
    exists(obj | ((obj.OclIsTypeOf(IndirectActorObj)
      or obj.OclIsTypeOf(DirectActorObj))
      and obj.actor.isPrimaryActor()))
  )
)
or
(sentence.predicate.complements->size() != 0
and sentence.predicate.complements->
  exists(complt | complt.OclIsTypeOf(ObjPPComplt)
    and complt.toActor->size() = 1
    and complt.toActor.isPrimaryActor())
)
)
```

isPrimaryActor() is used to check whether the actor is the primary actor of the use case specification.

Figure 72 Formal specification/identification of Response2PrimaryActor

Context Sentence

```
sentence.functionType.OclIsTypeOf(ActionSentence) and
self.transactionType.OclIsTypeOf(Reponse2SecondaryActor) implies
self.OclAsTypeOf(SimpleSentence).subject.OclIsTypeOf(SysSubj)
and
(
  (sentence.predicate.objects->size() != 0 and sentence.predicate.objects->
    exists(obj | ((obj.OclIsTypeOf(IndirectActorObj)
      or obj.OclIsTypeOf(DirectActorObj))
      and obj.actor.isSecondaryActor()))
  )
)
or
(sentence.predicate.complements->size() != 0
and sentence.predicate.complements->
  exists(complt | complt.OclIsTypeOf(ObjPPComplt)
    and complt.toActor->size() = 1
    and complt.toActor.isSecondaryActor())
)
)
```

isSecondaryActor() is used to check whether the actor is one of the secondary actors of the use case specification.

Figure 73 Formal specification/identification of Response2SecondaryActor

Appendix E FormalizeUCM – Algorithm of

ParsingSimpleSentence

```

SimpleSentence parse_SimpleSentence(String sentence)
{
    Subject subj_model = null
    Predicate pred_model = null
    AdvModifier advModifier_model = null
    Predicator predicator_model = null

    public Collection<TypedDependency> tdl = null
    String sentence_type = null
    Tree sentence_tree = null
    Tree predicate_tree = null
    Tree predicator = null
    Tree subj_tree = null
    Tree subj_head = null
    Tree ADVP_tree = null

    sentence_tree = parseSentence2Tree(Sentence)
    tdl = obtainDependencies(sentence_tree)

    FOR each children tree (child_tree) of sentence_tree
        IF child_tree.OclIsTypeOf (ADVP)
            ADVP_tree = child_tree
            advModifier_model = parse_ADVP(ADVP_tree)
        ENDIF
        IF child_tree.OclIsTypeOf(VP)
            predicate_tree = child_tree
            sentence_type = identifySentenceType(predicate_tree, tdl)
            Predicator_model = parsePredicator(predicate_tree, sentence_type)
            Predicate_model = parsePredicate(predicate_tree, Predicator_model, sentence_type)
            //use dependency "subj" identify the head of the subject_tree
            subj_head = identifySubject_head(predicate_tree, tdl)
        ENDIF
    ENDFOR

    subj_tree = identifySubject_tree(sentence_tree, subj_head)
    subj_model = parseSubject(subj_tree, sentence_type)
    return createSimpleSentence(subj_model, predicate_model, advModifier_model)
}

String identifySentenceType(Tree predicate_tree, tdl){
    String sentence_type = 'normal'
    FOR each dependency (dep) of tdl
        IF dep == 'cop' && dep.dep() == predicate_tree.getHeadOfTree()
            sentence_type = "copula"
        ENDIF
        IF dep == 'aux' && dep.dep() == predicate_tree.getHeadOfTree().OclIsTypeOf('VBG')
            sentence_type = 'present participle'
        ENDIF
    ENDIF
}

```

```

ENDIF
IF dep == 'aux' && dep.dep() == predicate_tree.getHeadOfTree().OclIsTypeOf('VBN')
    sentence_type = 'past participle'
ENDIF
IF dep == 'auxpass'
    sentence_type = 'passive'
ENDIF
IF dep == 'expl'
    sentence_type = 'there-be sentence'
ENDIF
ENDFOR
return sentence_type
}

```

```

Predicator parsePredicator(Tree predicate_tree, sentence_type){
    SWITCH sentence_type
        case 'normal'
            predicator_tree = predicate_tree.getHeadOfTree()
        case 'copula'
            predicator_tree = dep.OclAsTypeOf('cop').dep()
        case 'present participle'
            predicator_tree = dep.OclAsTypeOf('aux').gov()
        case 'past participle'
            predicator_tree = dep.OclAsTypeOf('aux').gov()
        case 'passive'
            predicator_tree = dep.OclAsTypeOf('auxpass').gov()
        case 'there-be sentence'
            predicator_tree = dep.OclAsTypeOf('expl').gov()
    return createPredicator(predicator_tree)
}

```

```

Predicate parsePredicate(Tree predicate_tree, Predicator predicator_model, sentence_type){
    DirectObject dobj_model = null
    IndirectObject iobj_model = null
    SubjectComplt subjComplt_model = null
    List<ObjectComplt> objComplts_model = new ArrayList<ObjectComplt>()

    FOR each dependency (dep) of tdl
        IF dep == 'dobj' && dep.gov == predicator_model
            dobj_model = parseDirectObject(dep);
            dobj_tree = predicate_tree.getChildTreeContaining(dep.dep());
            IF dobj_tree == 'NP'
                IF dobj_tree.npPoHS.OclIsTypeOf(PrepositionalPhrase)
                    objComplts_model.add(parseObjPPComplt(dobj_tree))
                IF dobj_tree.npPoHS.OclIsTypeOf(InfinitivePhrase)
                    objComplts_model.add(parseObjInfinitiveComplt(dobj_tree))
                IF dobj_tree.npPoHS.OclIsTypeOf(PresentParticiple)
                    objComplt_model.add(parseObjIngParticiplePhrase(dobj_tree))
                IF dobj_tree.npPoHS.OclIsTypeOf(PastParticiple)
                    objComplt_model.add(parseObjEdParticiplePhrase(dobj_tree))
            IF dep == 'iobj' && dep.gov == predicator_model
                idobj_model = parseIndirectObject(dep)
            IF sentence_type == 'copula' && dep == 'cop'

```

```

        subjComplt_model = parseSubjectCompletment(predicate_tree, tdl)
    IF sentence_type == 'passive' && dep == 'agent'
        objComplts_model.add(parseObjPPComplt(dep))
    IF sentence_type == 'there-be sentence' && dep == 'nsubj'
        thereBeSentence_model = parseThereBeSentence(dep)
    IF dep == 'acompl' && dep.gov = predicator_tree
        objComplt_model.add(parseObjAdjPComplt(dep))
    IF dep == 'xcomp' && dep.gov = predicator_tree
        objComplt_model.add(parseObjInfinitiveComplt(dep))
    IF dep == ('prep' || 'prepc') && dep.gov = predicator_tree
        objComplt_model.add(parseObjPPComplt(dep))
    IF dep == 'purpcl' && dep.gov = predicator_tree
        objCmplt_model.add(parseObjInfinitiveComplt(dep))
    EndFor
    return createPredicate(predicator_model, dobj_model, iobj_model, subjComplt_model,
objComplts_model)
}

Subject parseSubject(Tree subj_tree, String sentenceType){
    if subj_tree.OcIsTypeOf(NounPhrase)
        return createSubject(parseNounPhrase(subj_tree))
    if subj_tree.OcIsTypeOf(InfinitivePhrase)
        return createSubject(parseInfinitivePhrase(subj_tree))
    if subj_tree.OcIsTypeOf(GerundPhrase)
        return createSubject(parseGerundPhrase(subj_tree))
}

```

Appendix F Automatically generated class diagrams

In this section, we present the class diagrams automatically generated by aToucan for the following case study systems: ATM (Appendix F.1), ARENA (Appendix F.2), Elevator (Appendix F.3), Video Store (Appendix F.4), Cab Dispatching (Appendix F.5), Payroll (Appendix F.6), and Car Parts Dealer (Appendix F.7) systems. The characteristics of each case study system are discussed in Chapter 13.

F.1 ATM System

The automatically generated class diagram for the ATM system is presented in **Figure 74** to **Figure 79**.

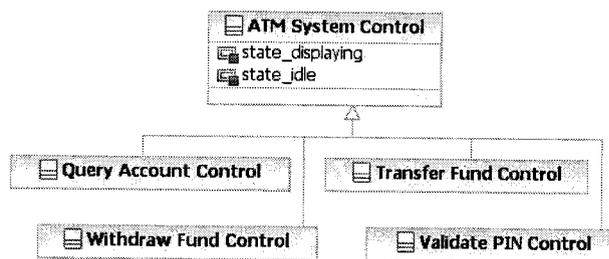


Figure 74 ATM System - control classes

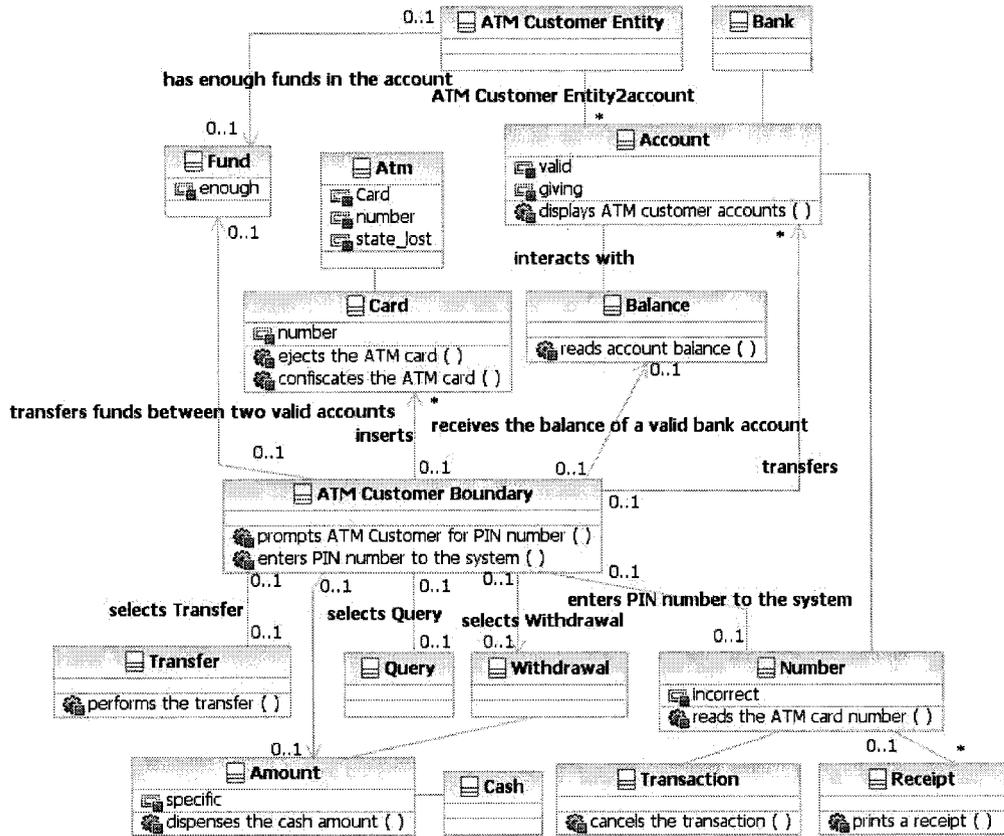


Figure 75 ATM System - entity classes

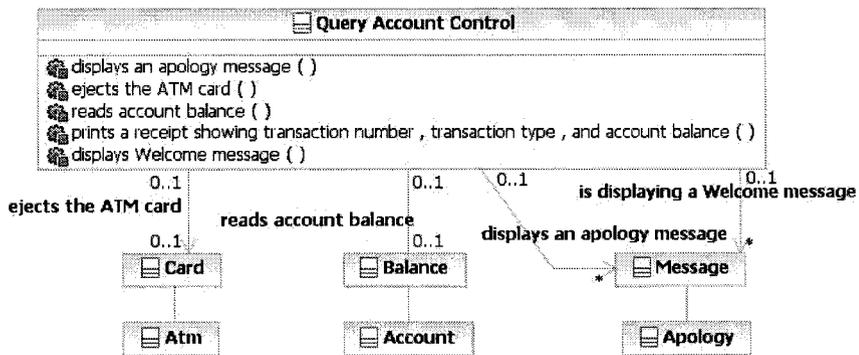


Figure 76 ATM System - Query Account Control class

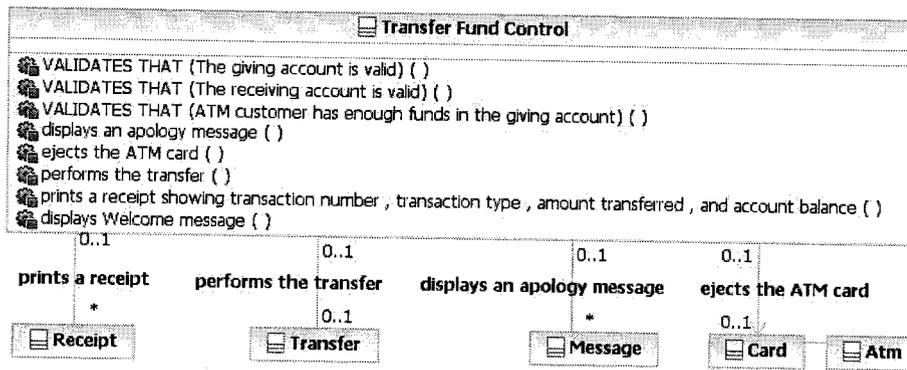


Figure 77 ATM System - Transfer Fund Control class

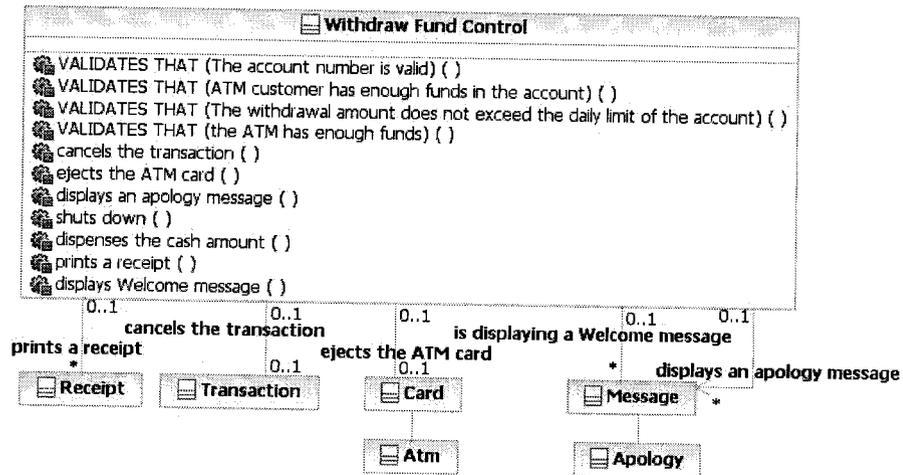


Figure 78 ATM System - Withdraw Fund Control class

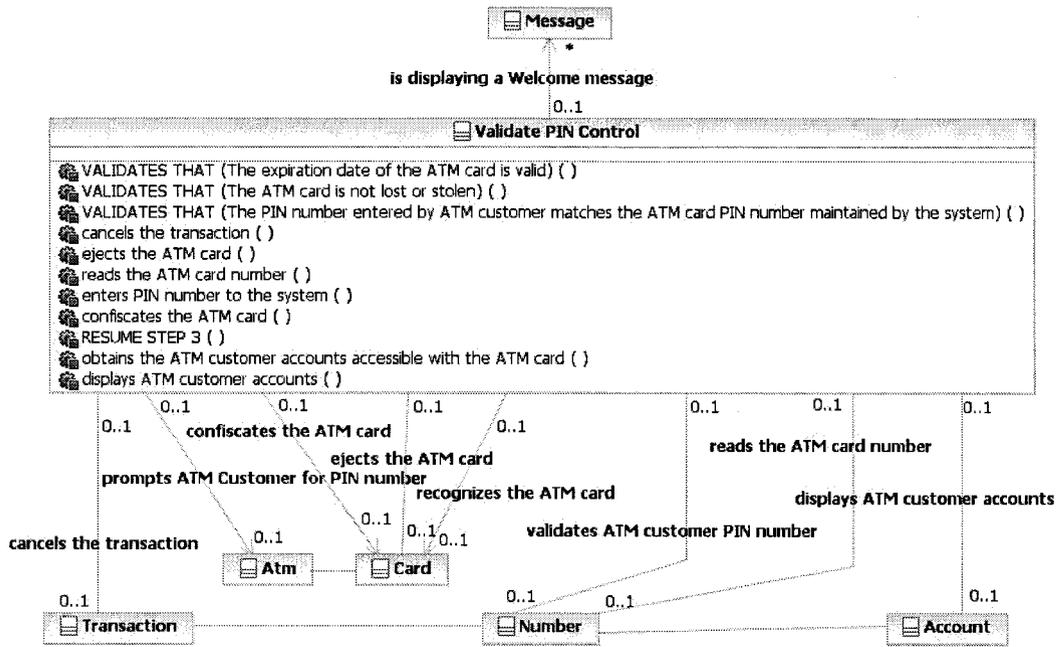


Figure 79 ATM System - Validate PIN Control class

F.2 ARENA system

The automatically generated class diagram for the ARENA system is presented in Figure 80 and Figure 81. Notice that this class diagram is derived from a single use case description called *Announce Tournament*. This perhaps explains the reason why there are some isolated classes, as shown in Figure 80; the use case model is not complete so that aToucan is not able to generate some associations to connect these isolated classes with the rest of the class diagrams.

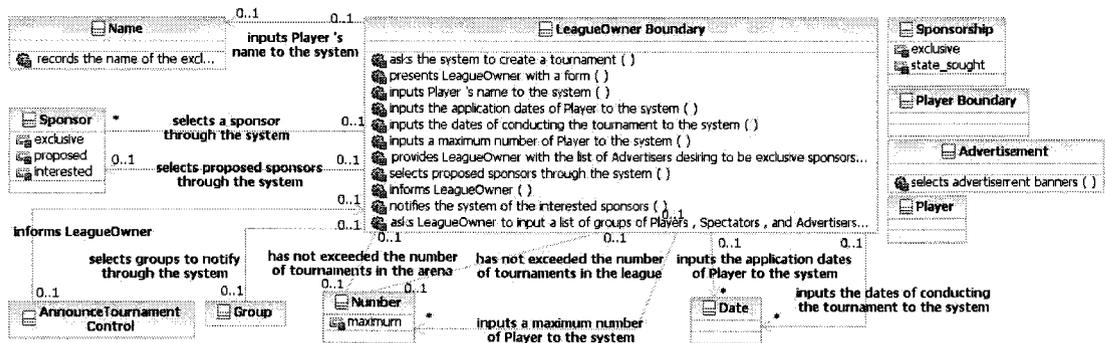


Figure 80 ARENA system – part 1

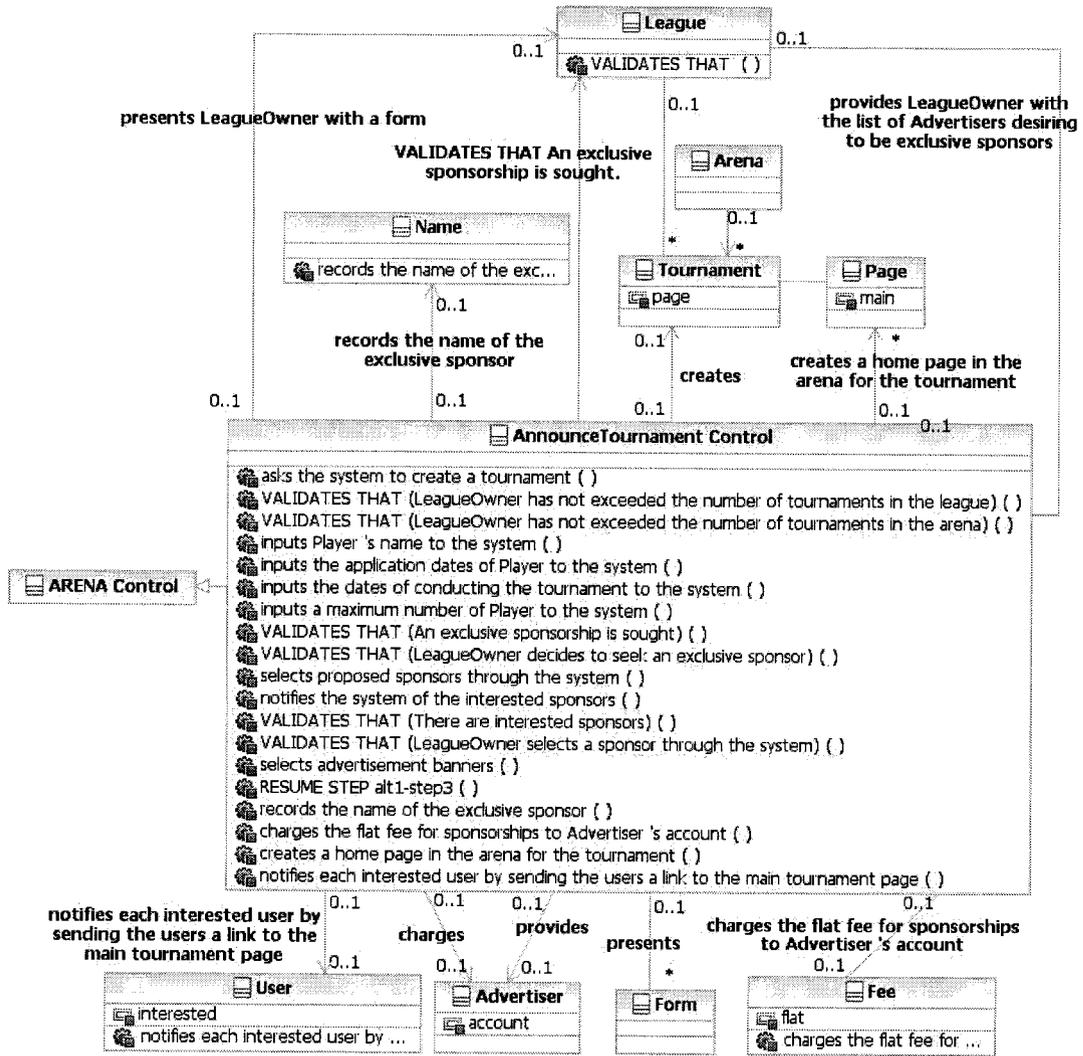


Figure 81 ARENA system – part 2

F.3 Elevator system

The main part of the automatically generated class diagram for the Elevator system is presented in Figure 82, Figure 83, and Figure 84.

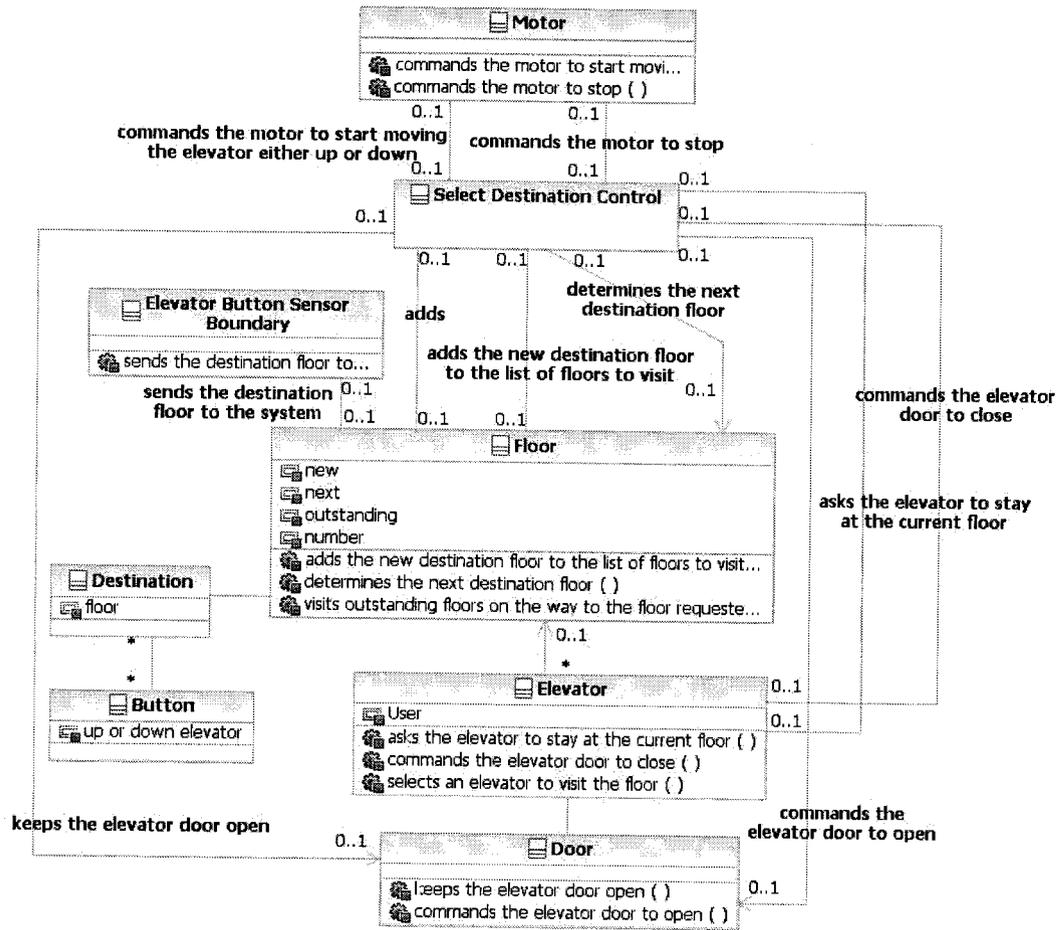


Figure 82 Elevator system – entity classes

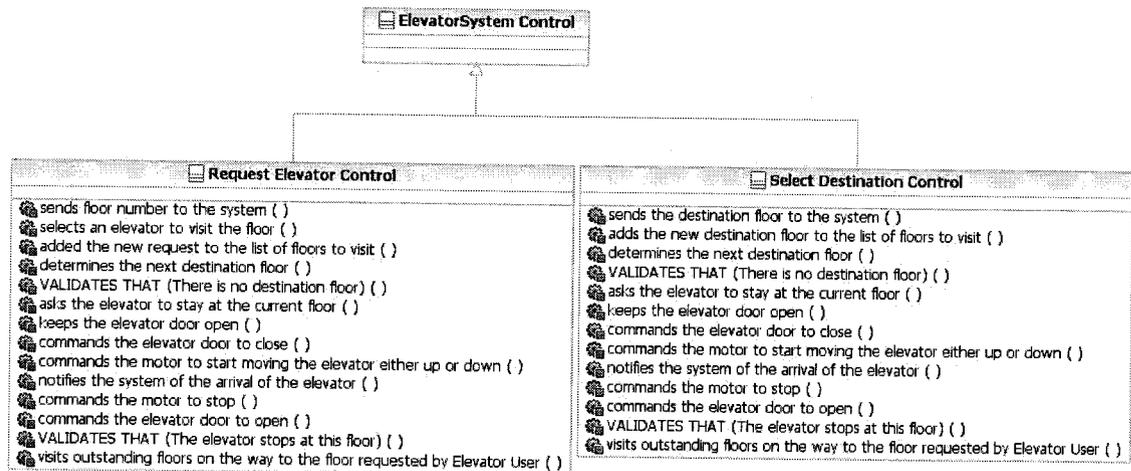


Figure 83 Elevator system – control classes

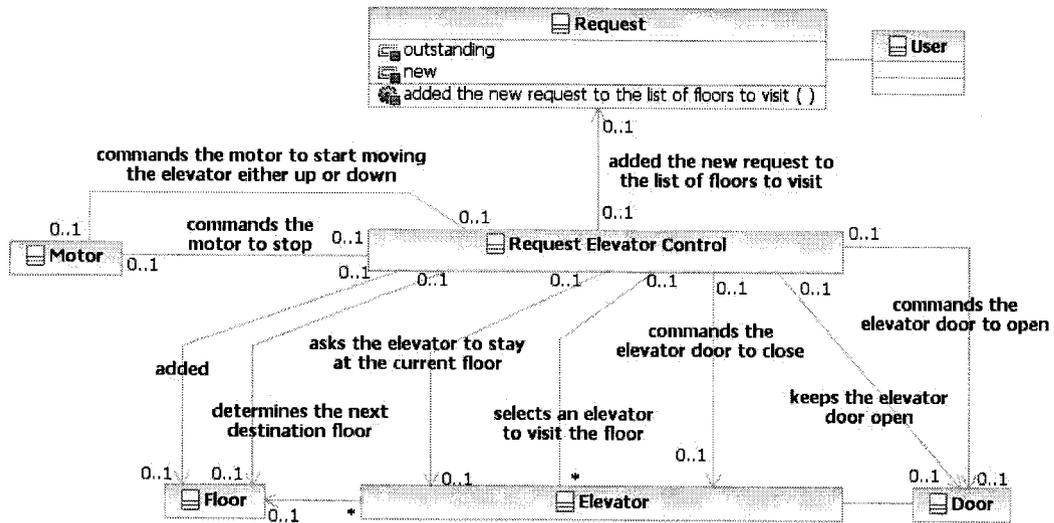


Figure 84 Elevator system – Request Elevator Control

F.4 Video Store system

The automatically generated class diagram for the Video Store system is presented in Figure 85 to Figure 92.

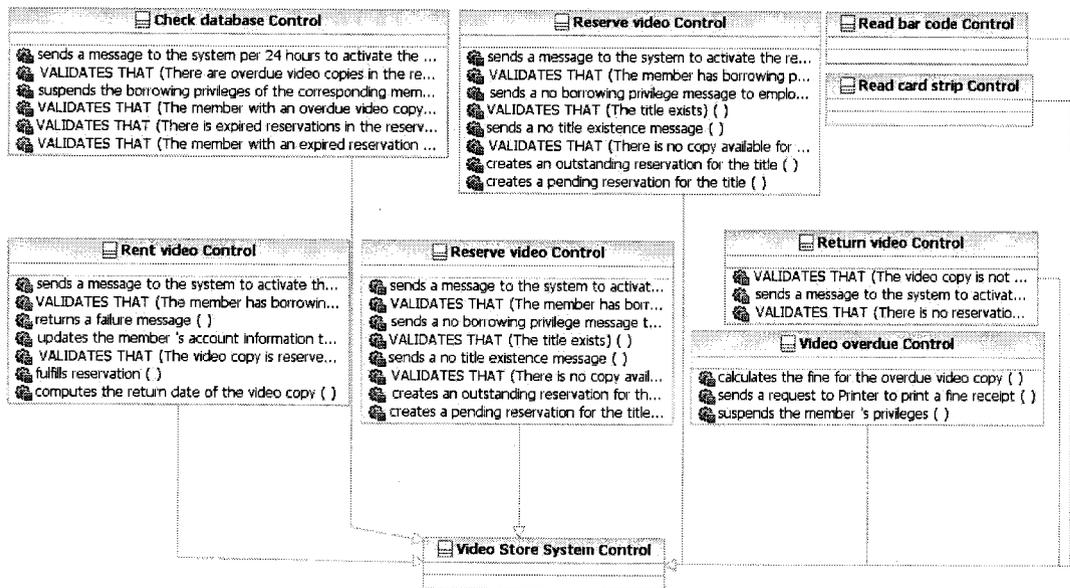


Figure 85 Video Store system – control classes

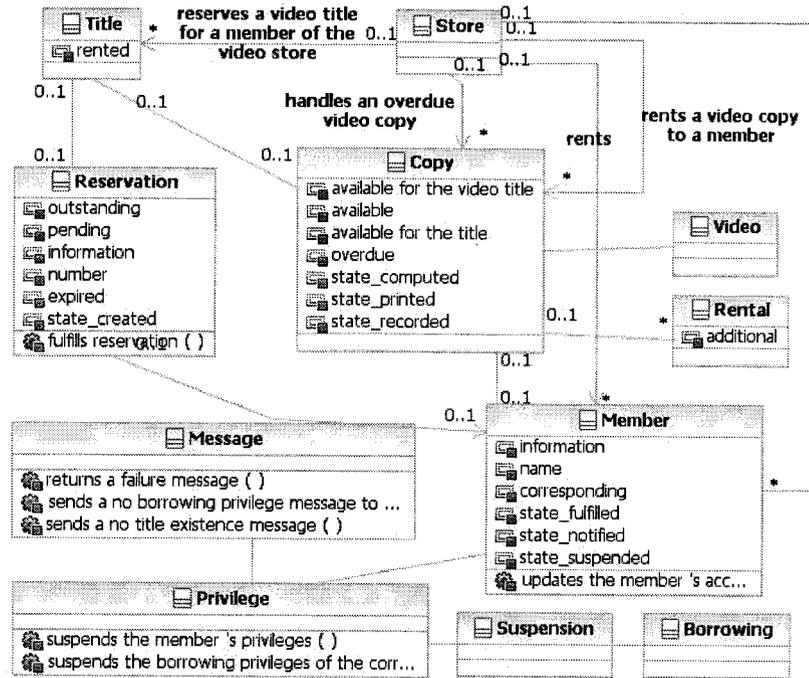


Figure 86 Video Store system – entity classes

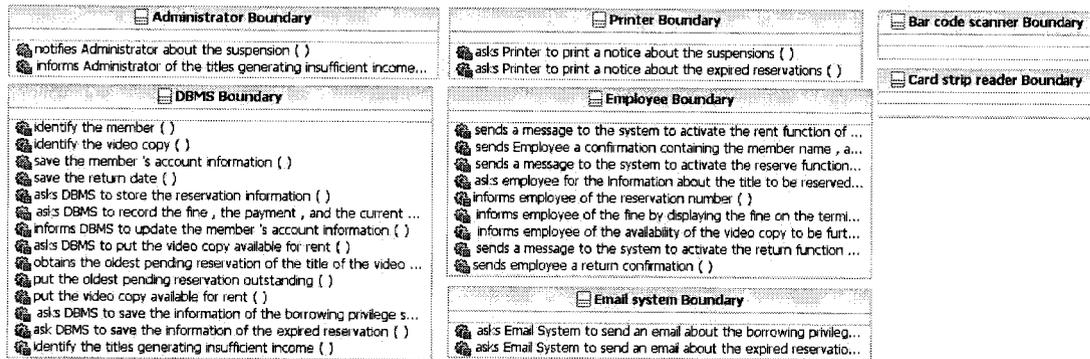


Figure 87 Video Store system – boundary classes

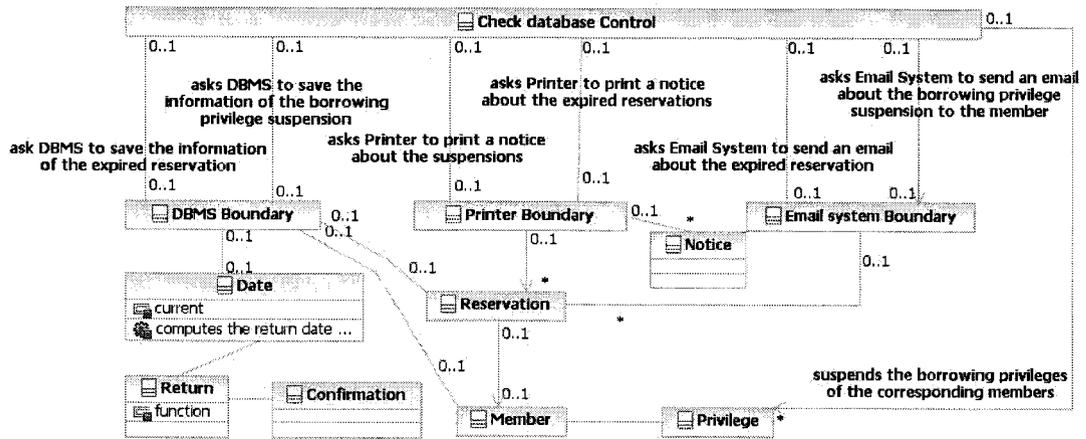


Figure 88 Video Store system – Check database Control

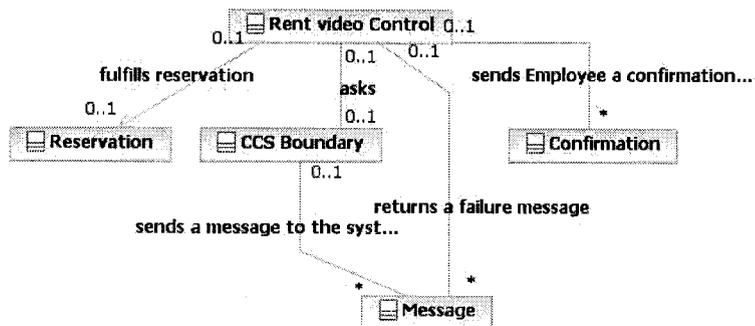


Figure 89 Video Store system – Rent video Control

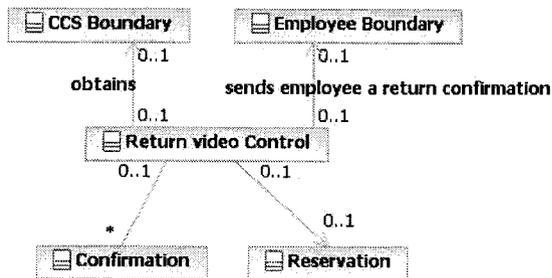


Figure 90 Video Store system –Return video Control

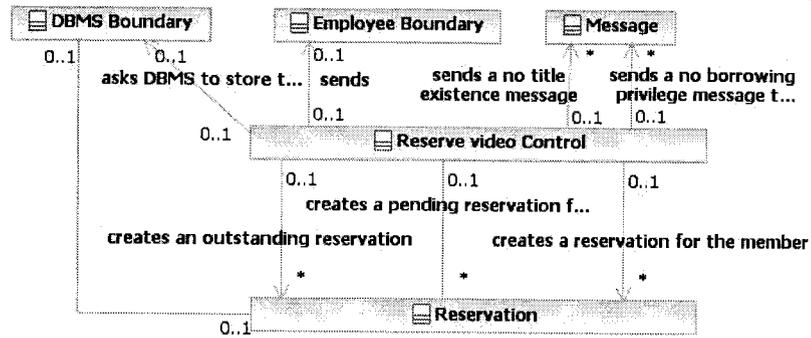


Figure 91 Video Store system – Return video Control

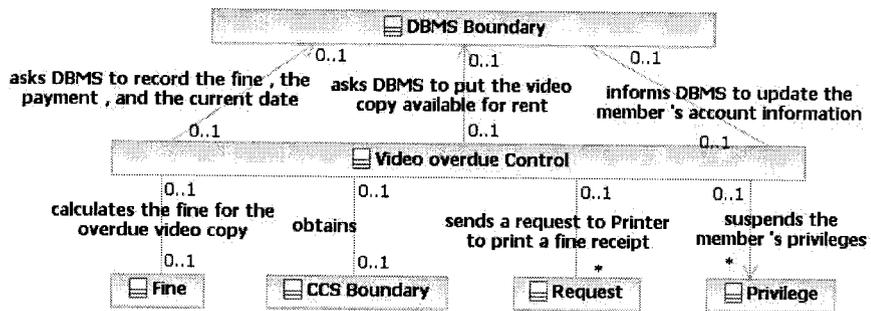


Figure 92 Video Store system – Video overdue Control

F.5 Cab Dispatching system

The main part of the automatically generated class diagram for the Cab Dispatching system is presented in Figure 93, Figure 94, and Figure 95.

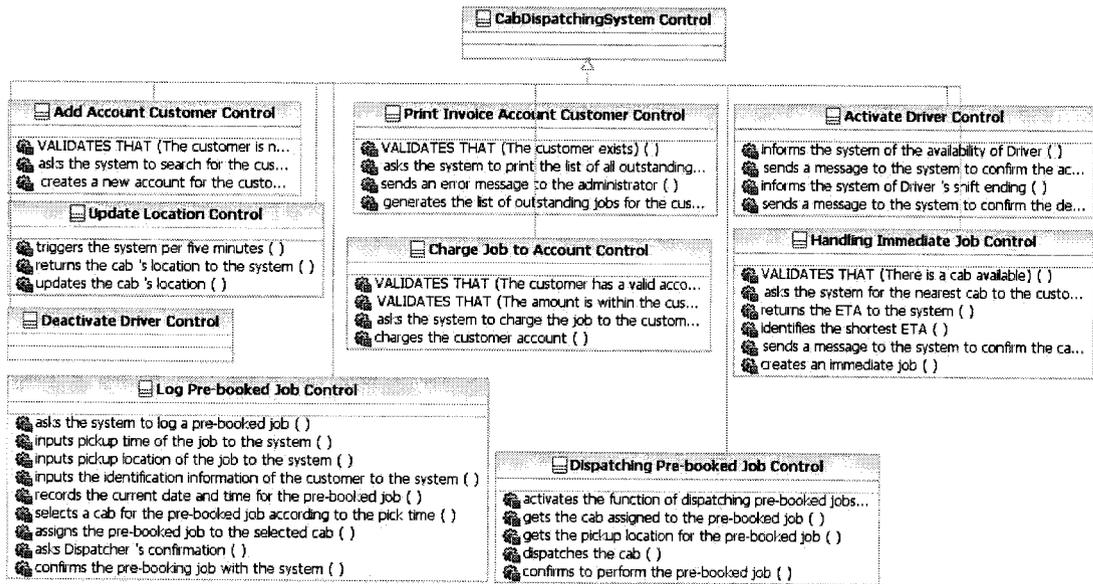


Figure 93 Cab Dispatching system – control classes

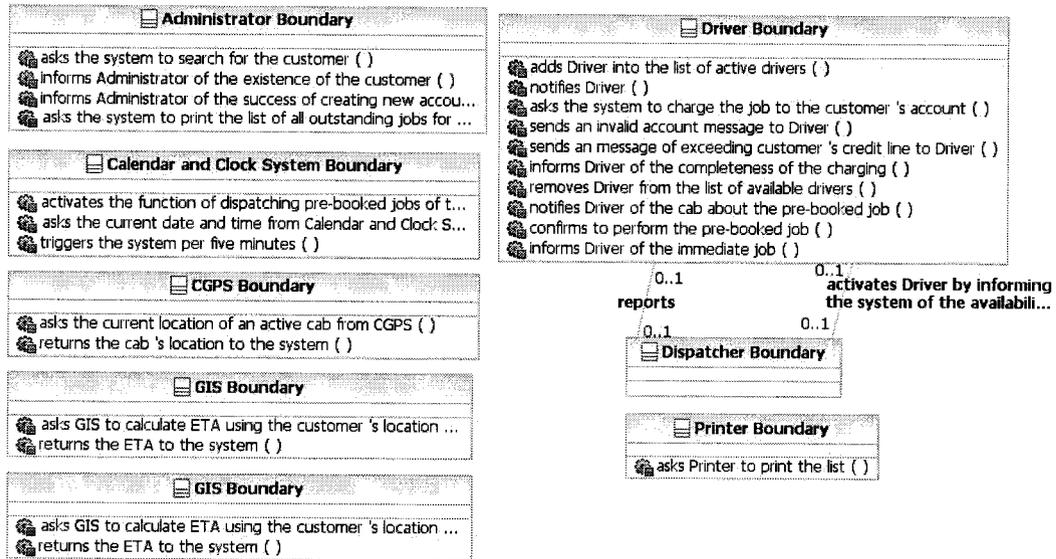


Figure 94 Cab Dispatching system – entity classes

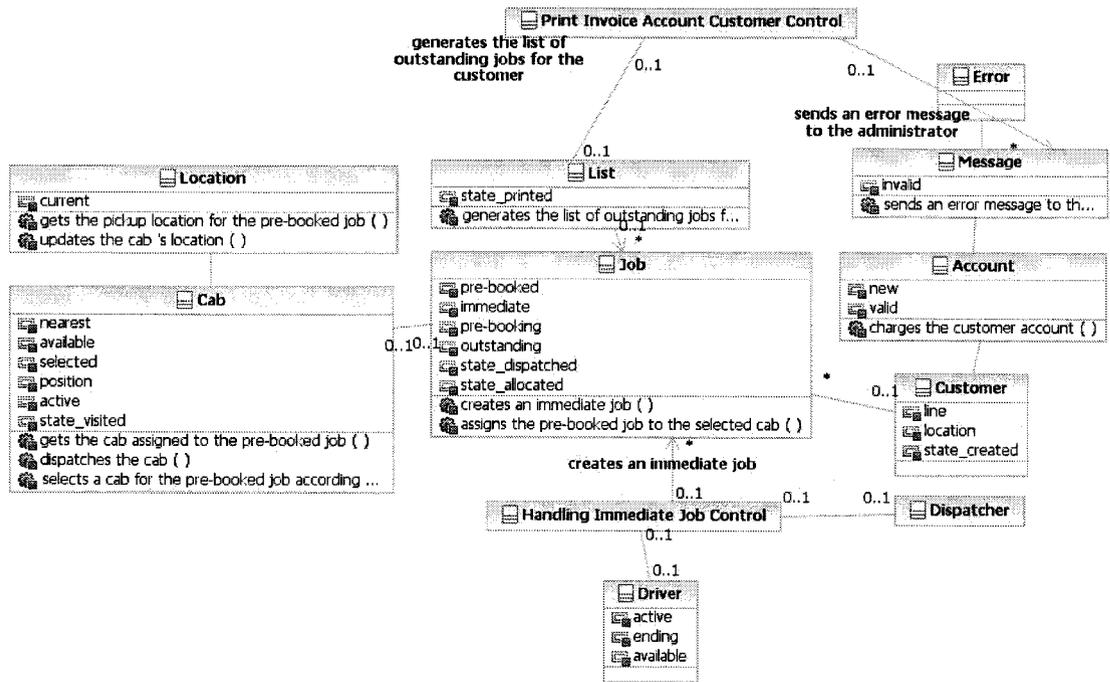


Figure 95 Cab Dispatching system – entity classes

F.6 Payroll system

The main part of the automatically generated class diagram for the Payroll system is presented in Figure 96, Figure 97, and Figure 98.

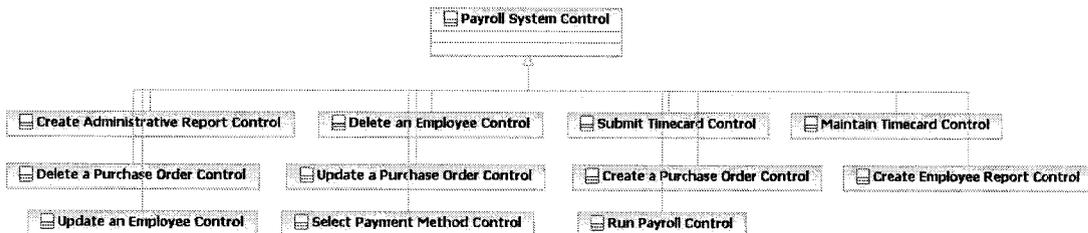


Figure 96 Payroll system – control classes

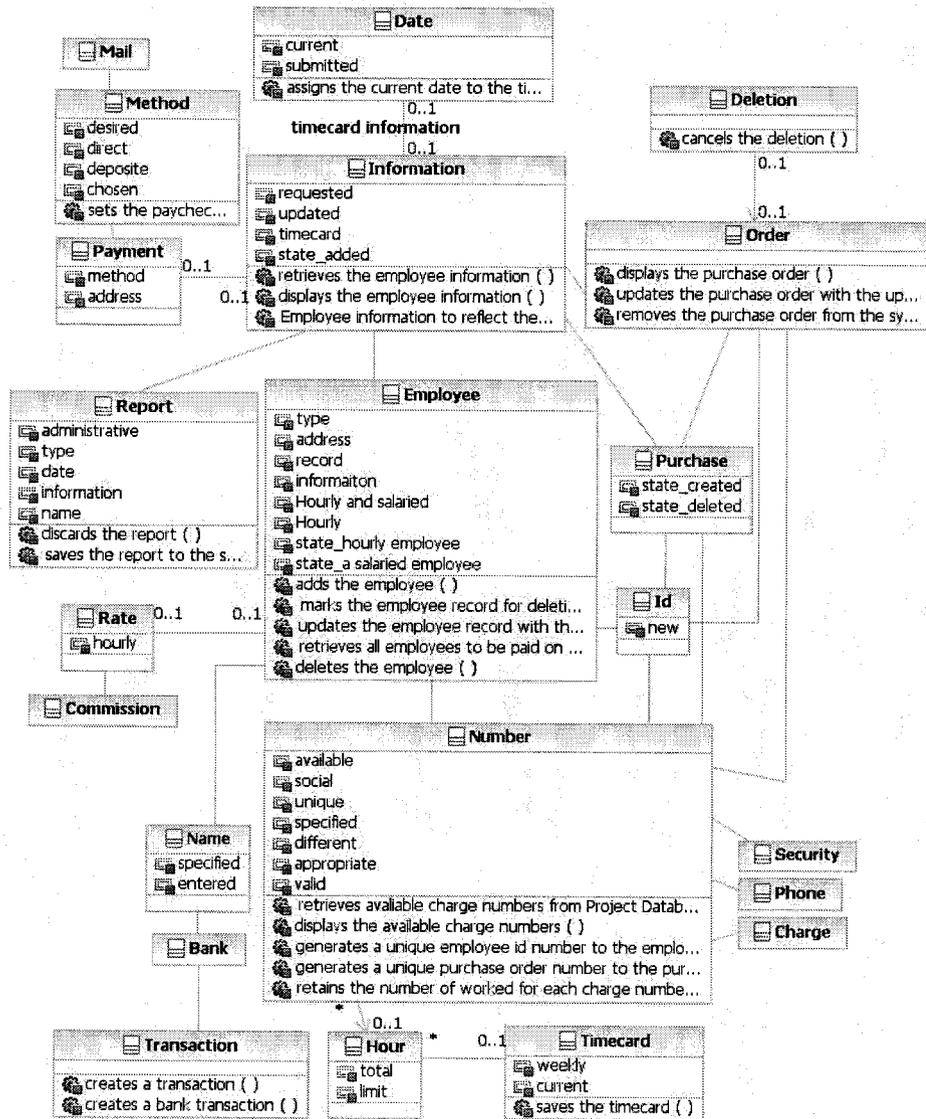


Figure 97 Payroll system – entity classes

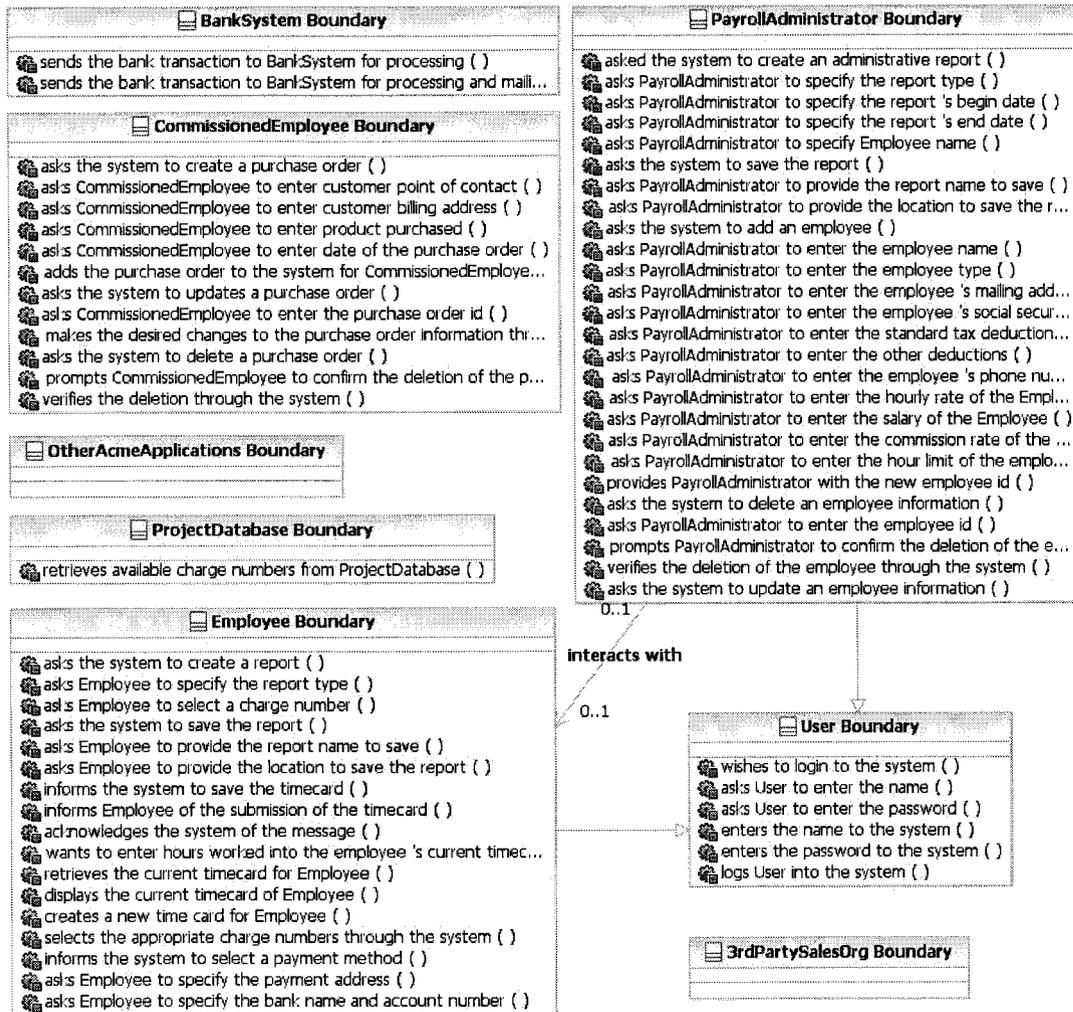


Figure 98 Payroll system – boundary classes

F.7 Car Parts Dealer (CPD) system

The entity classes the automatically generated class diagram for the CPD system is provided in Figure 99.

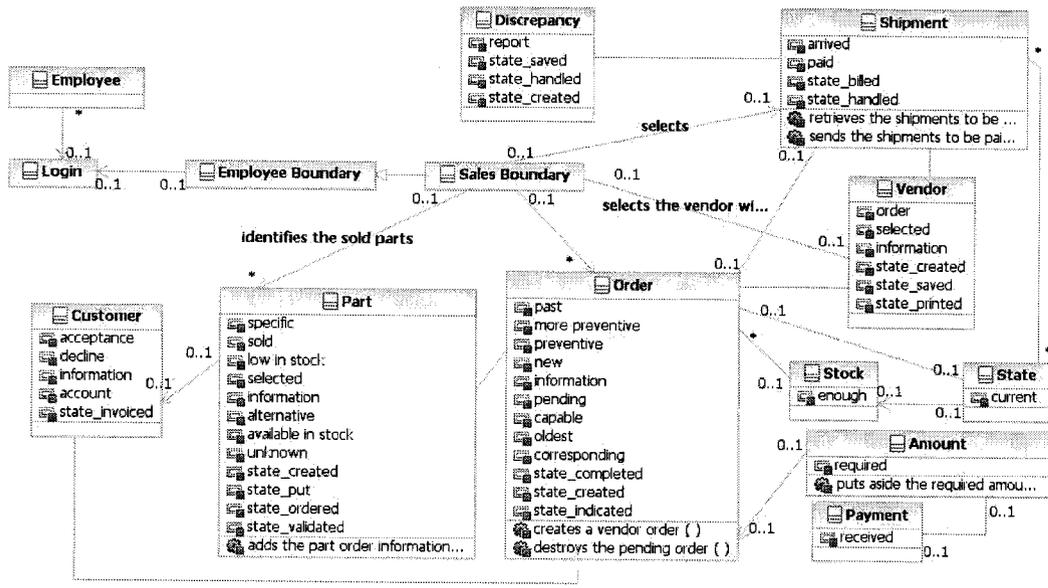


Figure 99 CPD system – entity classes

Appendix G Automatically generated sequence diagrams

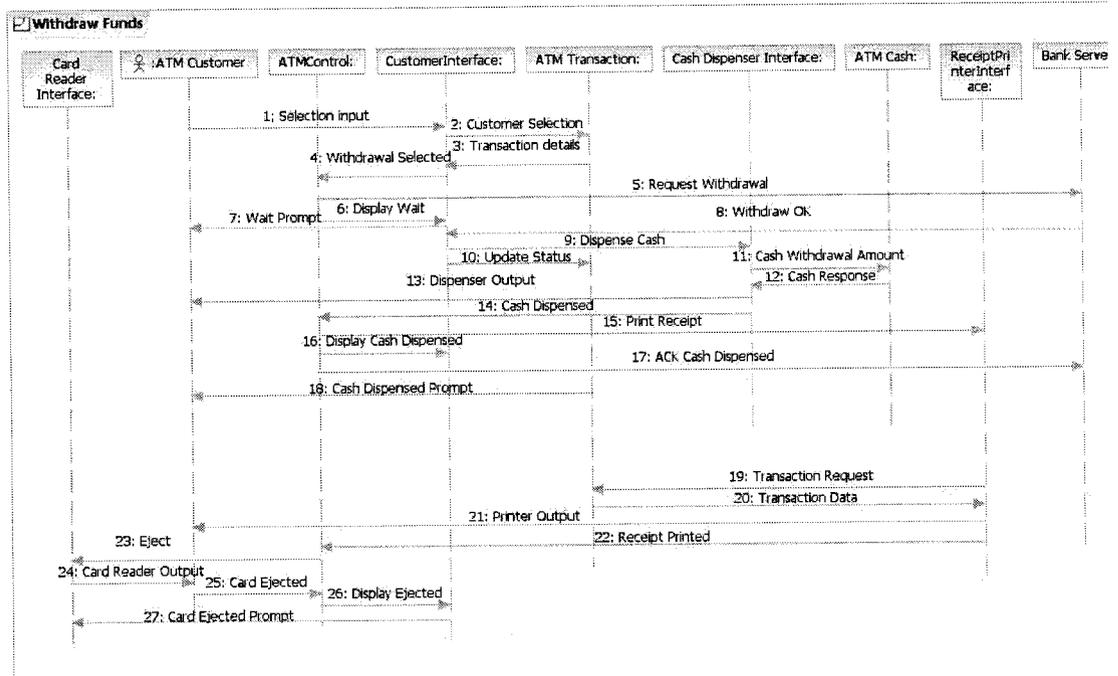


Figure 100 ATM System – Withdraw Funds (sequence diagram in the textbook)

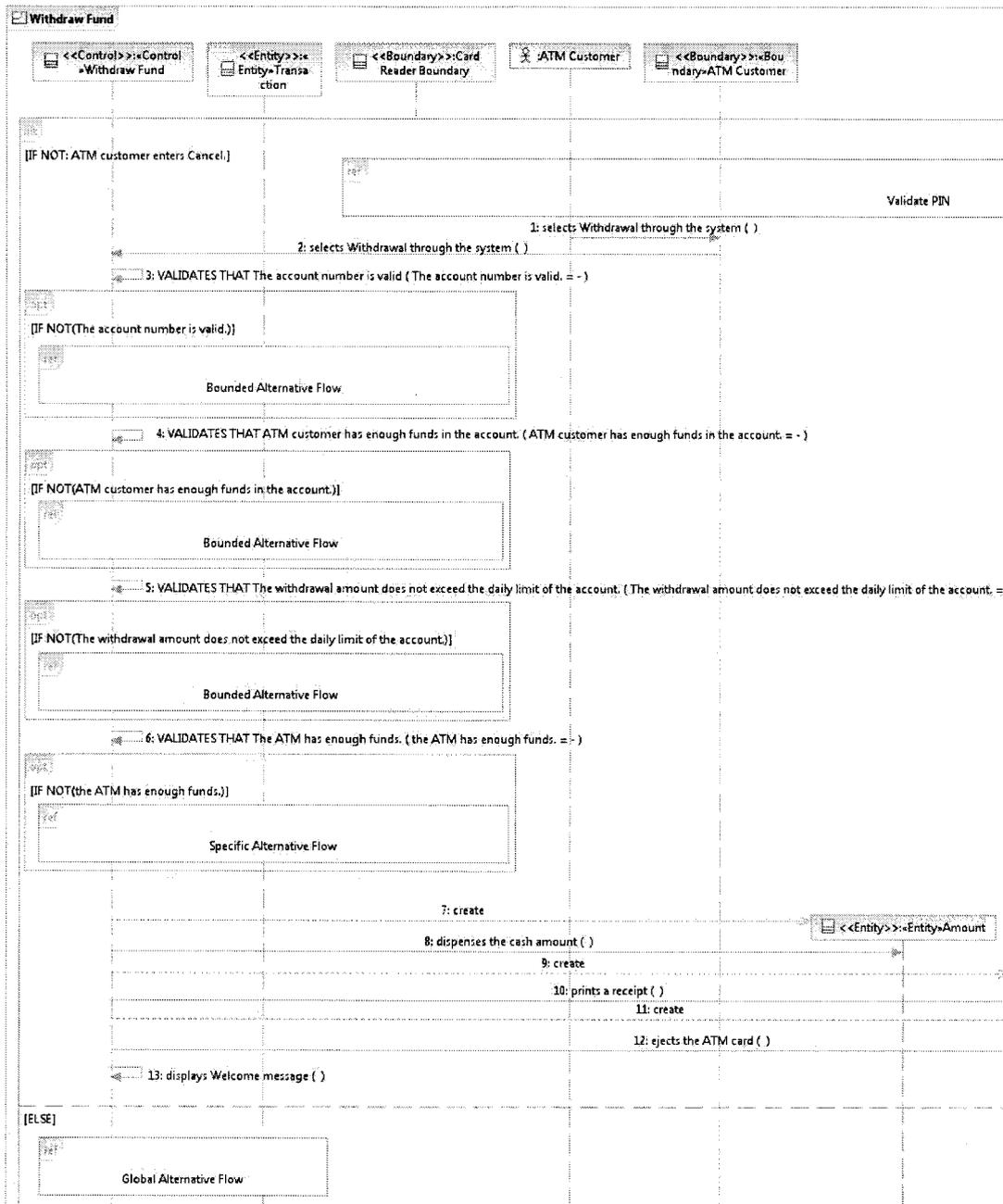


Figure 101 Sequence diagram generated for the basic flow of use case Withdraw Fund (in part)

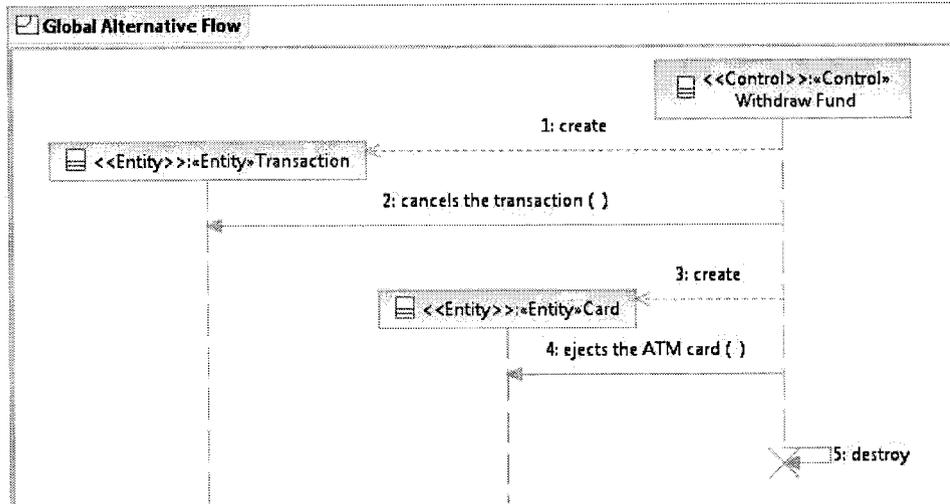


Figure 102 Sequence diagram generated for the global alternative flow of use case Withdraw Funds

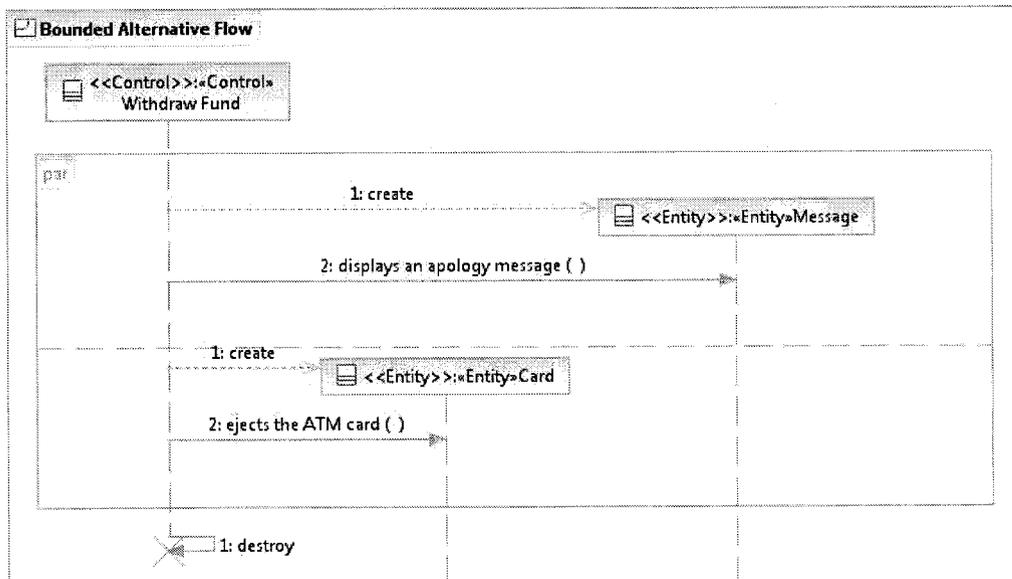


Figure 103 Sequence diagram generated for the bounded alternative flow of use case Withdraw Funds

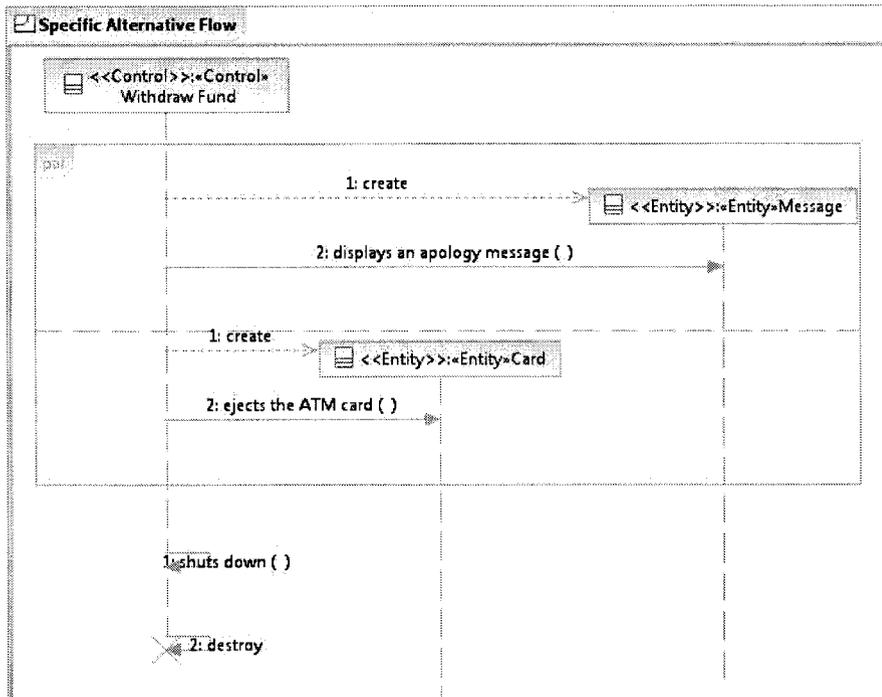


Figure 104 Sequence diagram generated for the specific alternative flow of use case Withdraw Funds

Appendix H UCSs and automatically generated activity diagrams

In this appendix, we provide the re-written UCSs and automatically generated activity diagrams of aToucan and the three commercial tools compared with aToucan: Visual Paradigm, CaseComplete, and Ravenflow.

H.1 aToucan

The re-written UCS by applying RUCM is provided in Table 10. The automatically generated overview and detailed activity diagrams are presented in Figure 105 and Figure 106, respectively. Data flow information for the detailed activity diagram is given in Table 59-not directly attached to it to avoid overcrowding the diagram.

As shown in Table 59, no input pins were generated for action 12 and d. The sentence corresponding to action d is "The system shuts down." Therefore the action has no input pin. Because NL parser cannot correctly parse the sentence corresponding to action 12: "The system displays Welcome message". So this is an error. An input pin should be generated for the action.

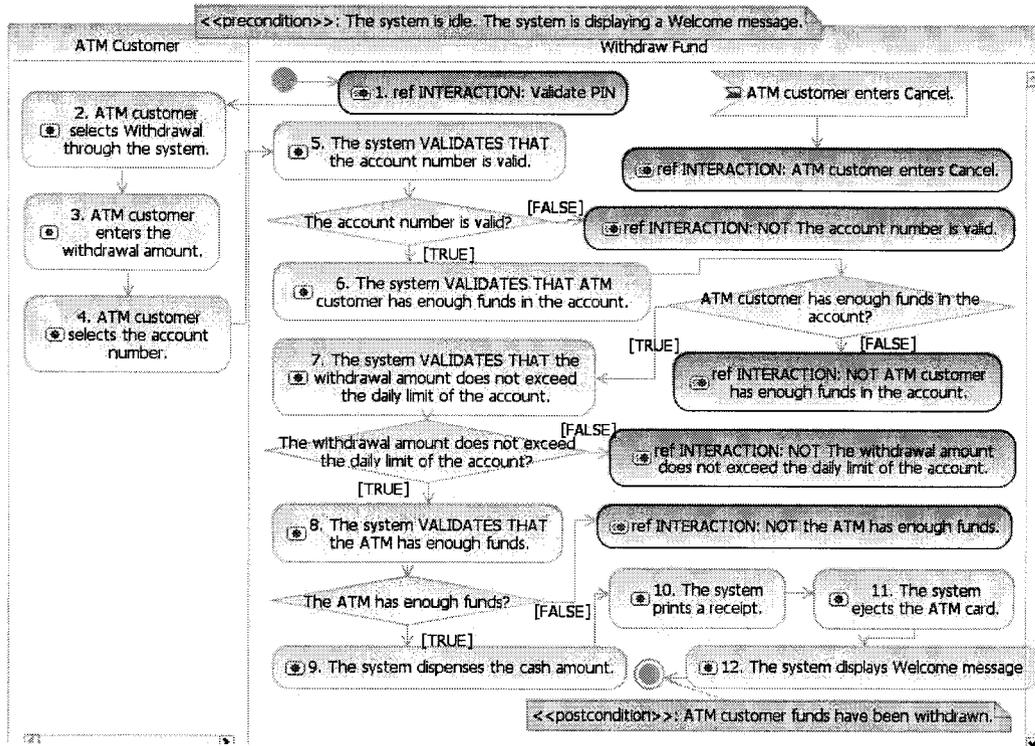


Figure 105 Overview Activity Diagram – aToucan

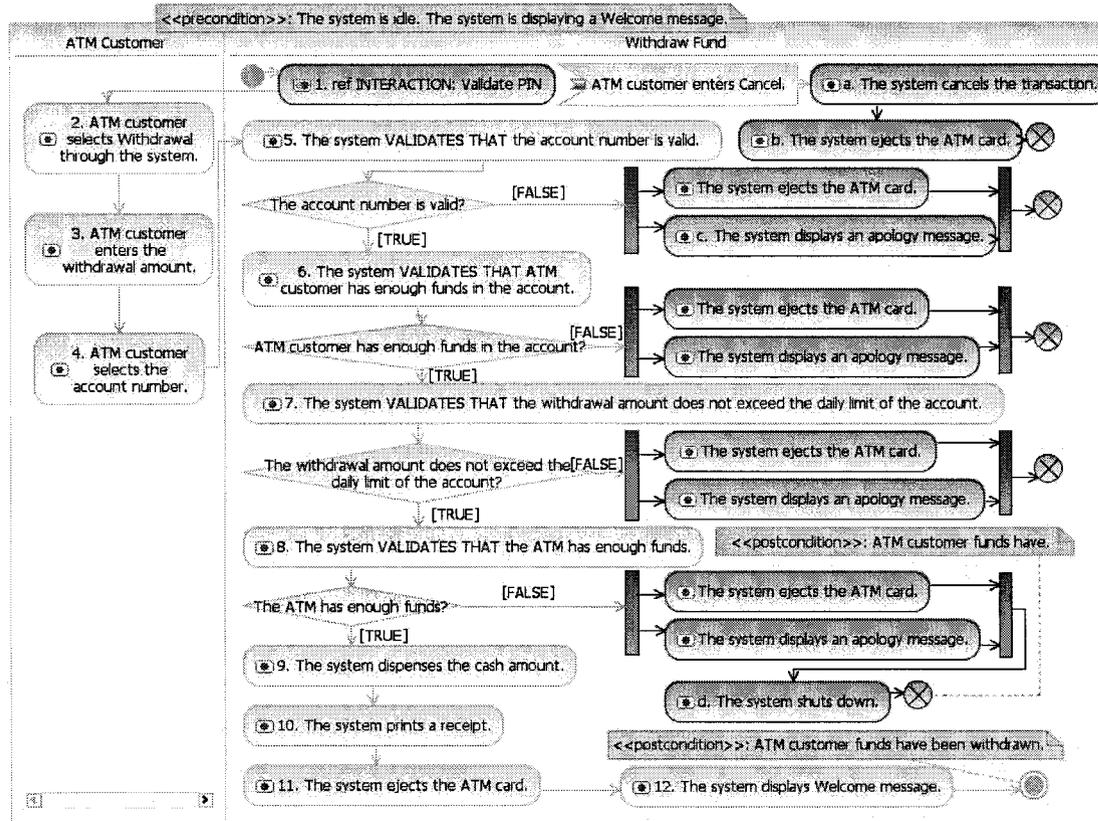


Figure 106 Detailed Activity Diagram – aToucan

Table 59 Data flow informatin on detailed activity diagram

Action	Typed class(es) of pins
2 (out)	Withdrawal
3 (out)	Amount
4 (out)	Account Number
5 (in)	Account number
6 (in)	Customer, Fund, Account
7 (in)	Amount
8 (in)	ATM, Fund
9 (in)	Amount
10 (in)	Receipt
11 (in)	Card
12 (in)	--
a (in)	Transaction
b (in)	Card
c (in)	Message
d (in)	--

H.2 Visual Paradigm

The re-written UCS according to the format enforced by Visual Paradigm is provided in Figure 107. The automatically generated activity diagram is presented in Figure 108.

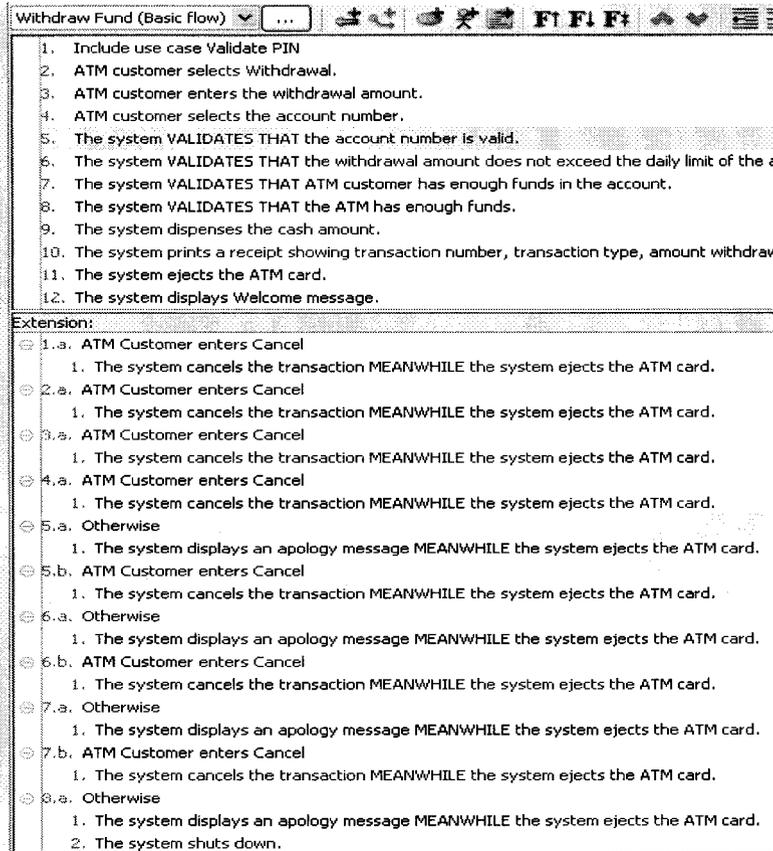


Figure 107 Re-written UCS - Visual Paradigm (partial)

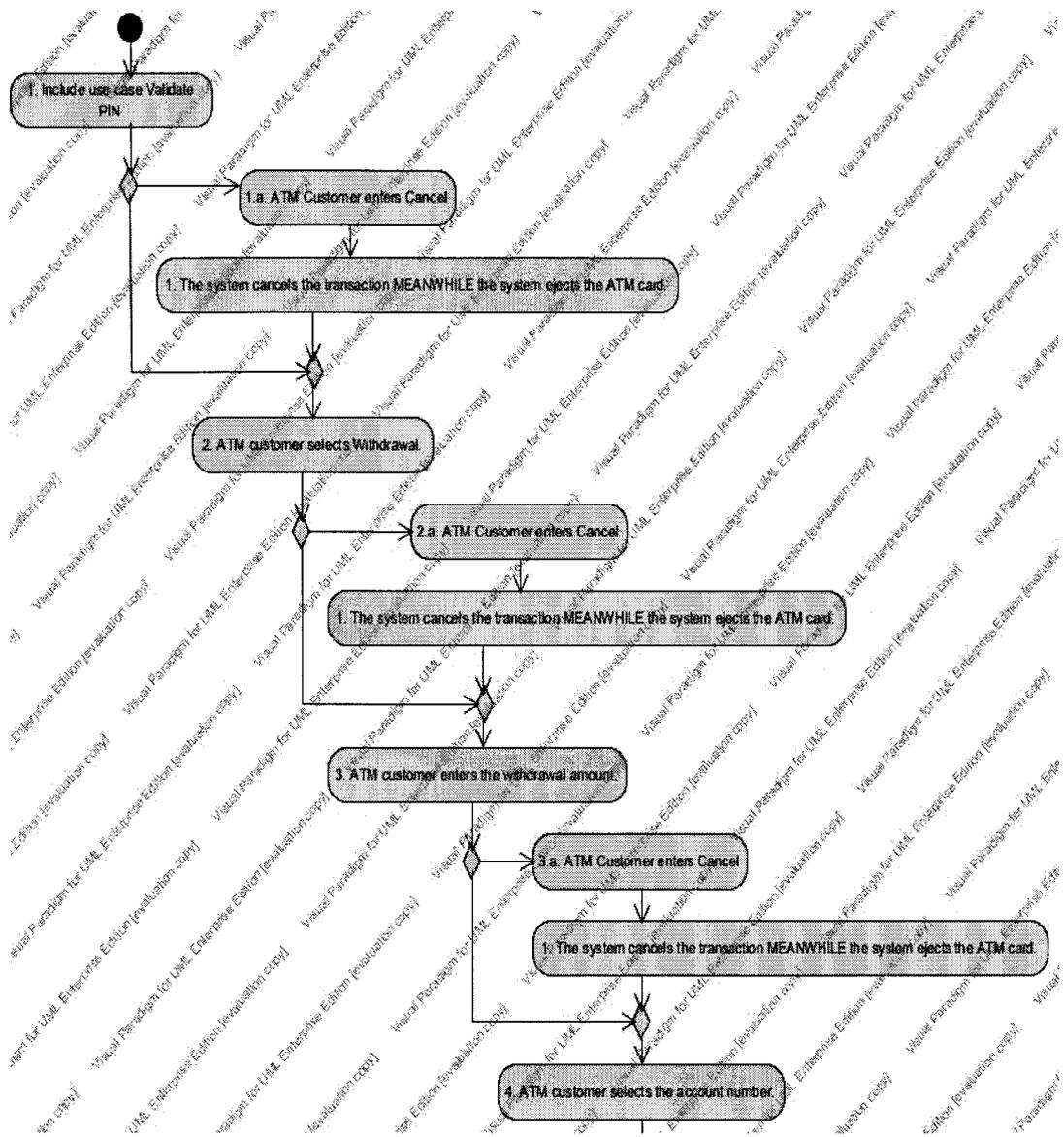


Figure 108 Activity Diagram - Visual Paradigm (partial)

H.3 Ravenflow

The re-written UCS according to the format enforced by Ravenflow is provided in Figure 110. The automatically generated activity diagram is presented in Figure 109. Notice that there is data flow information was generated for the activity diagram.

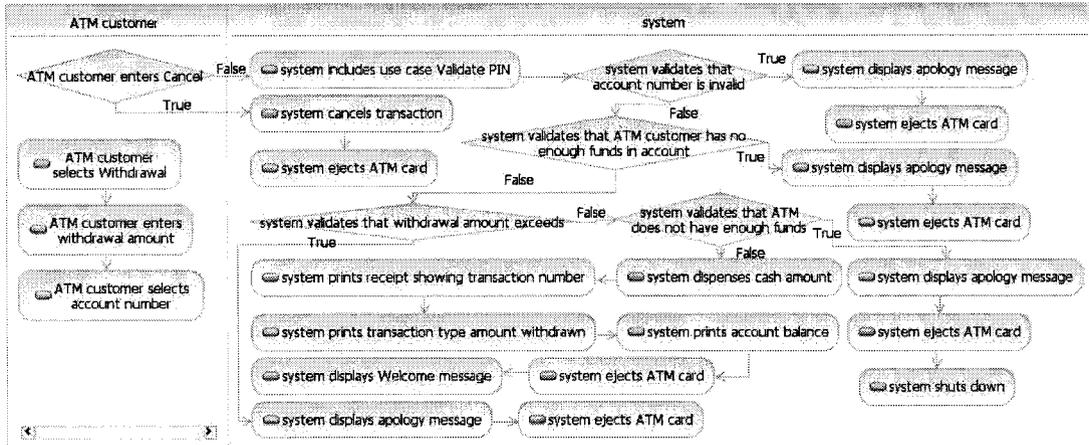


Figure 109 Activity Diagram - Ravenflow (for better layout, the activity diagram is exported to RSA and visualized by it)

-----Use Case-----

If ATM customer enters Cancel, then the system cancels the transaction meanwhile the system ejects the ATM card! Otherwise the system includes use case Validate PIN. ATM customer selects Withdrawal. ATM customer enters the withdrawal amount. ATM customer selects the account number. If the system validates that the account number is invalid, then the system displays an apology message and the system ejects the ATM card! Otherwise, if the system validates that ATM customer has no enough funds in the account, then the system displays an apology message meanwhile the system ejects the ATM card! Otherwise, if the system validates that the withdrawal amount exceeds, the system displays an apology message and the system ejects the ATM card! Otherwise, if the system validates that the ATM does not have enough funds, the system displays an apology message and the system ejects the ATM card, then the system shuts down! Otherwise, the system dispenses the cash amount. The system prints a receipt showing transaction number, transaction type, amount withdrawn, and account balance. The system ejects the ATM card. The system displays Welcome message.

Figure 110 Re-written UCS - Ravenflow

H.4 CaseComplete

The re-written UCS according to the format enforced by CaseComplete is provided in Figure 111. The automatically generated activity diagram is presented in Figure 112.

Steps Prose Expand Show Testing Procedure

Main Success Scenario:

1. Include use case Validate PIN
2. ATM customer selects Withdrawal.
3. ATM customer enters the withdrawal amount
4. ATM customer selects the account number.
5. The system VALIDATES THAT the account number is valid.
6. The system VALIDATES THAT ATM customer has enough funds in the account.
7. The system VALIDATES THAT the withdrawal amount does not exceed the daily limit of the account.
8. The system VALIDATES THAT the ATM has enough funds.
9. The system dispenses the cash amount.
10. The system prints a receipt showing transaction number, transaction type, amount withdrawn, and account balance.
11. The system ejects the ATM card.
12. The system displays Welcome message.

Extensions:

- 1.a. If ATM customer enters Cancel then
 1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
- 2.a. If ATM customer enters Cancel then
 1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
- 3.a. If ATM customer enters Cancel then
 1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
- 4.a. If ATM customer enters Cancel then
 1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
- 5.a. else
 1. The system displays an apology message MEANWHILE the system ejects the ATM card.
- 6.b. If ATM customer enters Cancel then
 1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
- 6.a. else
 1. The system displays an apology message MEANWHILE the system ejects the ATM card.
- 6.b. If ATM customer enters Cancel then
 1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
- 7.a. else
 1. The system displays an apology message MEANWHILE the system ejects the ATM card.
- 7.b. If ATM customer enters Cancel then
 1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
- 8.a. else
 1. The system displays an apology message MEANWHILE the system ejects the ATM card.
 2. The system shuts down.
- 8.b. If ATM customer enters Cancel then
 1. The system cancels the transaction MEANWHILE the system ejects the ATM card.
- 9.a. If ATM customer enters Cancel then
 1. The system cancels the transaction MEANWHILE the system ejects the ATM card.

Figure 111 Re-written UCS - CaseComplete (partial)

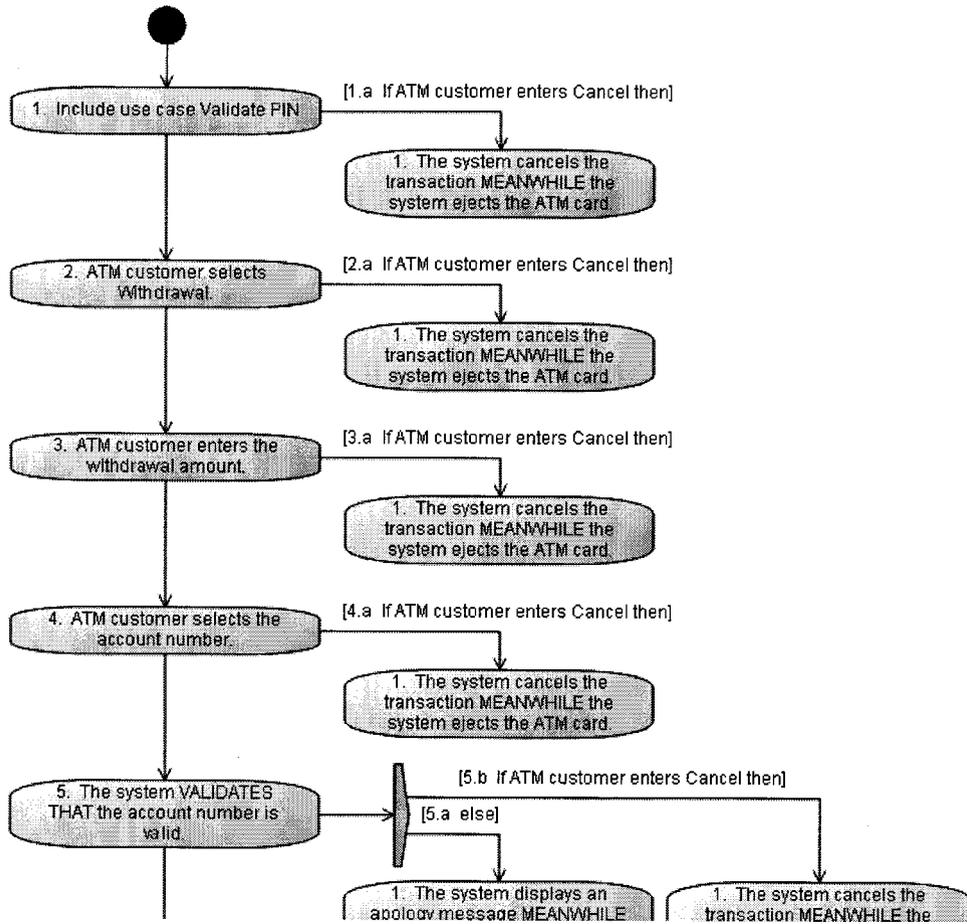


Figure 112. Activity Diagram - CaseComplete (partial)

Appendix I Experiment 1- Comprehension Questionnaire

Please put a (√) in the corresponding column.

You are strongly encouraged to refer to the list of restrictions you were provided with.
(The restriction numbers in the following tables match the numbers provided in the list of restriction specifications).

Table 60 Experiment 1 – Comprehension Questionnaire – Part 1 (R1-R16)

#	Restriction	I understood the restriction rule and was able to properly apply it.		The restriction rule is straightforward to apply.				Did you feel the restriction rule was too restrictive?					
		Yes	No	Completely agree	Generally agree	Generally disagree	Completely disagree	Completely agree	Generally agree	Generally disagree	Completely disagree		
1	The subjects of sentences in basic and alternative flows should be the system or actors.												
2	Describe the flow of events sequentially.												
3	Actor-to-actor interactions are not allowed.												
4	Describe one action per sentence.												
5	Use present tense only.												
6	Use active voice rather than passive voice.												
7	Clearly describe the interaction between the system and actors.												
8	Use declarative sentence only.												
9	Use words in a consistent way.												
10	Don't use modal verbs.												
11	Avoid adverbs.												
12	Use simple sentence only.												
13	Don't use negatively adverb and adjective.												
14	Don't use pronouns.												
15	Don't use participle phrases as adverbial modifier.												
16	Use "the system" to describe the system under design consistently.												

Table 61 Experiment 1 – Comprehension Questionnaire – Part 2 (R17-R26)

#	Restriction	I understood the restriction rule and was able to properly apply it.		The restriction rule is straightforward to apply.				Did you feel the restriction rule was too restrictive?					
		Yes	No	Completely agree	Generally agree	Generally disagree	Completely disagree	Completely agree	Generally agree	Generally disagree	Completely disagree		
17	INCLUDE USE CASE												
18	EXTENDED BY USE CASE												
19	RFS <basic flow step #>												
20	IF-THEN-ELSE-ENDIF (or ELSEIF-THEN-ENDIF)												
21	MEANWHILE												
22	VALIDATES THAT												
23	DO-UNTIL												
24	ABORT												
25	RESUME STEP												
26	Each basic flow and alternative flow should have their own postconditions.												

Appendix J Experiment 2 – Comprehension Questionnaire (CPD)

Instruction:

1. Please answer all questions, which are organized use case by use case.
2. For each multiple-choice question, please:
 - Select only one choice for each question, except those indicated with three asterisk (***)).

Please circle your answer.

- Justify your choice by further answering the question: “where did you get the information from”. Please clearly indicate the location (e.g., Basic flow – Step 2, Precondition, Postcondition) where you found information in the use case description to answer the multiple-choice question.
 - Some questions may have a choice: “Other answer”. If your answer is not in the provided choices, you may select this choice and provide your answer. Please also answer the question: “where did you get the information”.
3. A special question is asked for each use case. The following is the question:

*Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps of basic flow or alternative flows, and/or the subjects of actions? **YES or NO.***

If your answer is “yes”, please list the places and provide a brief explanation.

If you identify any confusion about the use case, please circle YES and describe the confusion.

4. Please make sure that your handwriting is recognizable.

Use case CreateCusotmerOrder

1. Which of the following scenarios is correct in terms of creating a new customer order?
 - a. Sales creates a new customer order without interacting with the system.
 - b. Sales requests the system to create a new customer order by providing the system the customer data.
 - c. The customer requests the system to create a new order for him/her. Then the system informs Sales the new created customer order.
 - d. The system creates a new customer order and informs Sales and the customer about the newly created customer order.
 - e. Not specified in the use case specification.
 - f. Other answers:

Where did you get the information from?

2. Which of the following scenarios is correct in terms of creating a part order for the customer?
 - a. Sales creates a part order indicated by the customer without interacting with the system.
 - b. Sales requests the system to create a part order.
 - c. The customer requests the system to create a new part order. Then the system informs Sales to create a new part order for the customer.
 - d. The system creates a part order and informs Sales and the customer about the newly created part order.
 - e. Not specified in the use case specification.
 - f. Other answers:

Where did you get the information from?

3. How is the newly created customer order saved?
 - a. The system saves the customer order by itself.
 - b. Sales keeps the customer order information.
 - c. The customer keeps his/her order information.
 - d. The system asks DBMS to save the customer order.
 - e. Not specified in the use case specification.
 - f. Other answers:

Where did you get the information from?

4. Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case specification, for example, in terms of the sequence of the steps of the basic flow or the alternative flows, the subjects of actions?

YES or NO.

If your answer is “yes”, please list the places and provide a brief explanation.

Use case OrderPart

5. How does Sales know whether the part number given by the customer is unknown or not?
- Sales has all the records of the part numbers, so he/she can figure out.
 - Sales directly asks DBMS to validate the part number.
 - Sales asks the system to validate the part number. Then the system retrieves the information from DBMS.
 - Not specified in the use case.
 - Other answers:

Where did you get the information from?

6. How does the system know whether there is enough stock for the customer part order or not?
- The system asks Sales for the information.
 - The system asks the customer for the information.
 - The system asks DBMS for the information.
 - The DBMS tells the system spontaneously.
 - Not specified in the use case.
 - Other answers:

Where did you get the information from?

7. When the part number is unknown, Sales provides an alternative part number to the system, then what happens?
- The system requests DBMS to check whether this alternative part number is known or unknown.
 - The system orders the part with the alternative part order for the customer.
 - The use case terminates.
 - Not specified in the use case.
 - Other answers:

Where did you get the information from?

8. Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps of the basic flow or the alternative flows, the subjects of actions?

YES or NO.

If your answer is “yes”, please list the places and provide a brief explanation.

Use case InsufficientStock

9. How does the system know whether the customer accepts the alternative part?
- Sales asks the customer whether an alternative part would do. Then Sales informs the system the customer’s decision.

- b. The customer tells the system his/her decision directly.
- c. The system does not need to know whether the customer accepts the alternative parts.
- d. Not specified in the use case
- e. Other answers:

Where did you get the information from?

10. Where is the part order information saved?

- a. The system
- b. Customer
- c. Sales
- d. DBMS
- e. Receiving&Shipping
- f. Accounting
- g. Not specified in the use case
- h. Other answers:

Where did you get the information from?

11. Who is responsible to create a pending order if the customer does not accept the alternative order provided by the system?

- a. The system
- b. Customer
- c. Sales
- d. DBMS
- e. Receiving&Shipping
- f. Accounting
- g. Not specified in the use case
- h. Other answers:

Where did you get the information from?

12. Where is the created pending order information saved if the customer does not accept the alternative order provided by the system?

- a. The system
- b. Customer
- c. Sales
- d. DBMS
- e. Receiving&Shipping
- f. Accounting
- g. Not specified in the use case
- h. Other answers:

Where did you get the information from?

13. Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps

of the basic flow or the alternative flows, the subjects of actions?

YES or NO.

If your answer is “yes”, please list the places and provide a brief explanation.

Use case CreateVendorOrder

14. Who is responsible to create a vendor order for the vendor selected by Sales, who has less number of discrepancies in the vendor’s previous shipments?
- The system
 - Customer
 - Sales
 - DBMS
 - Receiving&Shipping
 - Accounting
 - Not specified in the use case
 - Other answers:

Where did you get the information from?

15. Where is the created vendor order stored?
- The system
 - Customer
 - Sales
 - DBMS
 - Receiving&Shipping
 - Accounting
 - Not specified in the use case
 - Other answers:

Where did you get the information from?

16. Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps of the basic flow or the alternative flows, the subjects of actions?

YES or NO.

If your answer is “yes”, please list the places and provide a brief explanation.

Use case CompletePendingOrder

17. How does Sales obtain a pending order that can be completed when there are new arrivals from vendors?
- Sales asks customers.
 - Sales asks DBMS.

- c Sales asks the system. Then the system asks DBMS.
- d Sales maintains a record about the pending order.
- e Not specified in the use case
- f Other answers:

Where did you get the information from?

18. Where is the part order stored?
- a. The system
 - b. Customer
 - c. Sales
 - d. DBMS
 - e. Receiving&Shipping
 - f. Accounting
 - g. Not specified in the use case
 - h. Other answers:

Where did you get the information from?

19. Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps of the basic flow or the alternative flows, the subjects of actions?

YES or NO.

If your answer is “yes”, please list the places and provide a brief explanation.

Use case CreatePreventiveOrder

20. Where are the statistics on past orders stored?
- a. The system
 - b. Customer
 - c. Sales
 - d. DBMS
 - e. Receiving&Shipping
 - f. Accounting
 - g. Not specified in the use case
 - h. Other answers:

Where did you get the information from?

21. Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps of the basic flow or the alternative flows, the subjects of actions?

YES or NO.

If your answer is “yes”, please list the places and provide a brief explanation.

Appendix K Experiment 2 - Comprehension Questionnaire (VS)

Instruction:

1. Please answer all questions, which are organized use case by use case.
2. For each multiple-choice question, please:
 - Select only one choice for each question, except those indicated with three asterisk (***)).

Please circle your answer.

- Justify your choice by further answering the question: “where did you get the information from”. Please clearly indicate the location (e.g., Basic flow – Step 2, Precondition, Postcondition) where you found information in the use case to answer the multiple-choice question.
 - Some questions may have a choice: “Other answer”. If your answer is not in the provided choices, you may select this choice and provide your answer. Please also answer the question: “where did you get the information from”.
3. A special question is asked for each use case. The following is the question.

Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps of basic flow or alternative flows, and/or the subjects of actions? YES or NO.

If your answer is “yes”, please list the places and provide a brief explanation.

If you identify any confusion about the use case, please circle YES and describe the confusion.

4. Please make sure that your handwriting is recognizable.

Use case Rent Video

1. Where is the information of whether a member has borrowing privileges stored?
 - a. Administrator
 - b. Employee
 - c. The system itself.
 - d. DBMS
 - e. CCS
 - f. Read Card Strip
 - g. Not specified in the use case.
 - h. Other answers:

Where did you get the information from?

2. Which one of the following statements is correct?
 - a. First the system identifies the video copy from DBMS. Then the system updates the member's account information to reflect the additional rental.
 - b. First the system updates the member's account information to reflect the additional rental. Then the system identifies the video copy from DBMS.
 - c. The system identifies the video copy from DBMS, and concurrently the system updates the member's account information to reflect the additional rental.

Where did you get the information from?

3. How does the system check whether a video copy is reserved by a member?
 - a. The information is stored in the system itself.
 - b. The system gets the information by asking the employee.
 - c. The system requests the information from DBMS.
 - d. Not specified in the use case.
 - e. Other answers:

Where did you get the information from?

4. ***What information should have been saved into DBMS after the rental is completed? (you may want to select more than one choice)
 - a. The updated account information of the member.
 - b. The return date.
 - c. The member's name.
 - d. The member's identification number.
 - e. Not specified in the use case.
 - f. Other answers:

Where did you get the information from?

5. How does the employee know that the rental is confirmed?
 - a. The employee gets the confirmation from the member.
 - b. The employee gets the confirmation from the system.

- c. The employee gets the confirmation from DBMS.
- d. The employee gets the confirmation from another employee.
- e. Not specified in the use case.
- f. Other answers:

Where did you get the information from?

6. If the video copy is reserved by the member, then the reservation becomes fulfilled. What following action should the system take?
- a. The system exits.
 - b. The system sends the employee a confirmation.
 - c. The system communicates with DBMS to save the updated member's account information.
 - d. Not specified in the use case.
 - e. Other answers:

Where did you get the information from?

7. Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps of the basic flow or the alternative flows, the subjects of actions?

YES or NO.

If your answer is "yes", please list the places and provide a brief explanation.

Use case Return Video

8. Which one of the following statements is correct?
- a. First the system identifies the video copy from DBMS. Then the system checks whether the video copy is overdue.
 - b. First the system checks whether the video copy is overdue. Then the system identifies the video copy from DBMS.
 - c. The system identifies the video copy from DBMS, and concurrently the system checks whether the video copy is overdue.

Where did you get the information from?

9. How does the employee know that the video copy return is confirmed?
- a. The employee gets the confirmation from the member.
 - b. The employee gets the confirmation from the system.
 - c. The employee gets the confirmation from DBMS.
 - d. The employee gets the confirmation from another employee.
 - e. Not specified in the use case.
 - f. Other answers:

Where did you get the information from?

10. Where is the information regarding whether the title of the returning video copy is on hold (i.e., there is at least one reservation) by other members stored?
- The system itself
 - The employee
 - The member
 - The DBMS
 - The CCS
 - Not specified in the use case.
 - Other answers:

Where did you get the information from?

11. How does the system update the reservation status of a video copy from “on hold” to “outstanding”?
- The system itself updates the reservation status
 - Through the employee
 - Through the member
 - Through DBMS
 - Through CCS
 - Not specified in the use case.
 - Other answers:

Where did you get the information from?

12. In which condition, the use case Video Overdue is invoked? (If you cannot find the answer directly from the use case description, please indicate it.) (if you find the answer in the use case description, please indicate the place.)

13. Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps of the basic flow or the alternative flows, the subjects of actions?

YES or NO.

If your answer is “yes”, please list the places and provide a brief explanation.

Use case Video Overdue

14. Which one of the following statements regarding calculating the fine for the overdue video copy is correct?
- First, CCS sends the current date and time to the system. Then the system calculates the fine for the overdue video copy.
 - First, the system requests CCS to calculate the fine for overdue video copy. Then CCS returns the amount of fine to the system.
 - First, the system requests CCS to provide the current date and time. Then the system calculates the fine of the overdue video copy according to the current date and time returned by CCS.

- d. First, the system calculates the fine for the overdue video copy. Then the system requests the current date and time from CCS.

Where did you get the information from?

15. Who is responsible for requesting Printer to print the receipt for the fine?

- e. DBMS
- f. Employee
- g. The system
- h. The member (or customer)
- i. CCS
- j. Printer itself
- k. Not specified in the use case.
- l. Other answers:

Where did you get the information from?

16. What information should be stored into DBMS when the payment for the overdue video copy has been processed? (If you cannot find the answer directly from the use case description, please indicate it.) (if you find the answer in the use case description, please indicate the place.)

17. How does the employee know that the overdue video copy is available for further rental after the fine has been processed?

- a. The employee gets the confirmation from the member.
- b. The employee gets the confirmation from the system.
- c. The employee gets the confirmation from DBMS.
- d. The employee gets the confirmation from CCS.
- e. Not specified in the use case.
- f. Other answers:

Where did you get the information from?

18. How are the member's privileges suspended, when the member does not pay the fine?

- a. The employee suspends the member's privilege by telling the member the fact that his/her privileges are suspended.
- b. The employee informs the system that the member does not pay the fine. The system suspends the member's privileges and also informs DBMS to update the member's account information.
- c. Not specified in the use case.
- d. Other answers:

Where did you get the information from?

19. Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps of the basic flow or the alternative flows, the subjects of actions?

YES or NO.

If your answer is “yes”, please list the places and provide a brief explanation.

Use case Reserve Video

20. How does the system know whether a member has borrowing privileges before it reserves a video copy to the member?
- The information is stored in the system directly.
 - Inquiry DBMS.
 - The employee informs the system after asking the member either through email or telephone.
 - Not specified in the use case.
 - Other answers:

Where did you get the information from?

21. After the member who wants to make a reservation on a video title is identified, how does the system get the information regarding the title to be reserved?
- The information is stored in the system directly.
 - Inquiry DBMS.
 - The employee informs the system after asking the member.
 - The member tells the system directly.
 - Not specified in the use case.
 - Other answers:

Where did you get the information from?

22. Which one of the following statements is correct, when there is a copy available for the title that the member wants to reserve?
- First, the system creates an outstanding reservation for the title. Then the information of the outstanding reservation is stored into DBMS. At last, the reservation number is communicated to the member through employee.
 - First, DBMS creates an outstanding reservation for the title. Then DBMS informs the system that the reservation has been created along with the reservation number. At last, the reservation number is communicated to the member through employee.
 - First, the system creates an outstanding reservation for the title. Then the reservation number is communicated to the member through employee.
 - First, DBMS creates an outstanding reservation for the title. Then the reservation number is communicated to the member through employee.

Where did you get the information from?

23. What does the system do when it validates that the member has no borrowing privileges?
- The system asks for information regarding the title to be reserved.

- b. The system exits directly.
- c. The system informs the employee that the member has no borrowing privilege and then exits.
- d. They system makes a reservation for the member and informs employee about the created reservation.
- e. Not specified in the use case.
- f. Other answers:

Where did you get the information from?

24. What does the system do when it validates that the title the member requested to reserve does not exist?

- a. The system checks whether there are copies available for this title.
- b. The system exits directly.
- c. The system informs the employee that the title does not exist and then exits.
- d. They system makes a reservation for the member and informs employee about the created reservation.
- e. Not specified in the use case.
- f. Other answers:

Where did you get the information from?

25. Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps of the basic flow or the alternative flows, the subjects of actions?

YES or NO.

If your answer is “yes”, please list the places and provide a brief explanation.

Use case Check Database

26. Where is the information of rentals and overdue video copies stored?

- a. The information is stored in the system directly.
- b. Members of the store
- c. DBMS
- d. Employee
- e. CCS
- f. Administrator
- g. Email System
- h. Not specified in the use case.
- i. Other answers:

Where did you get the information from?

27. Where is the information of expired reservations stored?

- a. The information is stored in the system directly.

- b. Members of the store
- c. DBMS
- d. Employee
- e. CCS
- f. Administrator
- g. Email System
- h. Not specified in the use case.
- i. Other answers:

Where did you get the information from?

28. If there is no overdue video copies in rentals identified, what action(s) should the system take next?

- a. The system exits.
- b. The system continues to check whether there are any expired reservations.
- c. The system waits for employee's response.
- d. Not specified in the use case.
- e. Other answers:

Where did you get the information from?

29. If there is no expired reservation identified, what action(s) should the system take next?

- a. The system exits.
- b. The system continues to identify the titles generating insufficient income.
- c. The system waits for employee's response.
- d. Not specified in the use case.
- e. Other answers:

Where did you get the information from?

30. Did you identify any place(s) in the use case specification, which cause any confusion for you to understand the use case, for example, in terms of the sequence of the steps of the basic flow or the alternative flows, the subjects of actions?

YES or NO.

If your answer is "yes", please list the places and provide a brief explanation.

Appendix L Pre- and post- lab questionnaires

Pre-Lab Questionnaire (Experiment 1.1)

Levels of agreement:

1 – Completely agree

2 – Generally agree

3 – Generally disagree

4 – Completely disagree

Questions:

	1	2	3	4
• I have received extensive teaching on use case modeling.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• I have substantial experience with use case modeling.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• I am confident I can fully understand a use case diagram.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• I am confident I can accurately apply the use case template discussed in class.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• I fully understand the restrictions that apply to use case descriptions.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• I am confident I can accurately apply the restrictions.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Post-Lab Questionnaire (Experiment 1.1)

Levels of agreement:

1 – Completely agree

2 – Generally agree

3 – Generally disagree

4 – Completely disagree

Questions:

- | | 1 | 2 | 3 | 4 |
|---|--------------------------|--------------------------|--------------------------|--------------------------|
| 1. The instructions and objectives of the lab were perfectly clear to me. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| 2. I fully understood the use case diagram I was provided. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| 3. I was entirely comfortable when applying the use case template. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| 4. I was entirely comfortable when applying the restrictions. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| 5. I gained valuable experience by applying the restrictions
and in the future I will be in a better position to perform such tasks. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

6. Did you have enough time to finish the task? YES NO

If not, please answer the following sub-questions:

- | | | | | |
|---|--------------------------|--------------------------|--------------------------|--------------------------|
| • I spent too much time on understanding the case study system. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I spent too much time on understanding the restrictions. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I spent too much time on understanding the use case template. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Pre-Lab Questionnaire (Experiment 1.2)

Levels of agreement:

1 – Completely agree

2 – Generally agree

3 – Generally disagree

4 – Completely disagree

Questions:

- | | 1 | 2 | 3 | 4 |
|---|--------------------------|--------------------------|--------------------------|--------------------------|
| • I have good knowledge on UML modeling (class and sequence diagrams). | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I have substantial experience in building analysis models from use cases. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I am confident I can accurately apply the stereotypes <<Entity>> <<Boundary>>, and <<Control>> to classify classes. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I fully understand how to keep the consistency between class and sequence diagrams. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I am confident I can correctly use MS-Visio to draw class diagrams. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I am confident I can correctly use MS-Visio to draw sequence diagrams. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Post-Lab Questionnaire (Experiment 1.2)

Levels of agreement:

- 1 – Completely agree
- 2 – Generally agree
- 3 – Generally disagree
- 4 – Completely disagree

Questions:

	1	2	3	4
• The instructions and objectives of the lab were perfectly clear to me.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• I had plenty of time to finish the lab.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• I fully understood the use case descriptions I was provided.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• The sequence of the basic flow steps is very clear to me.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• The sequences of the alternative flow steps are very clear to me.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• The alternative flows clearly correspond to the basic flow.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• The use case template is very easy to apply.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• It was extremely hard for me to design the class diagram.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• It was extremely hard for me to design the sequence diagrams.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• It was extremely hard for me to keep the consistency between the class and the sequence diagrams.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• It is extremely hard for me use MS-Visio to draw class diagram.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
• It is extremely hard for me use MS-Visio to draw sequence diagrams.	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Pre-Lab Questionnaire (Lab 2.1 and 2.3)

Levels of agreement:

1 – Completely agree

2 – Generally agree

3 – Generally disagree

4 – Completely disagree

Questions:

- | | 1 | 2 | 3 | 4 |
|---|--------------------------|--------------------------|--------------------------|--------------------------|
| • I have good knowledge on UML class diagram modeling. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I have substantial experience in building class diagrams from use cases. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I am confident I can accurately apply the stereotypes <<Entity>> <<Boundary>>, and <<Control>> to classify classes. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I am confident I can correctly use MS-Visio to draw class diagrams. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Post-Lab Questionnaire (Lab 2.1 and 2.3)

Levels of agreement:

1 – Completely agree

2 – Generally agree

3 – Generally disagree

4 – Completely disagree

Questions:

- | | 1 | 2 | 3 | 4 |
|--|--------------------------|--------------------------|--------------------------|--------------------------|
| • The instructions and objectives of the lab were perfectly clear to me. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I had plenty of time to finish the lab. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I fully understood the use case descriptions I was provided. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • The sequence of the basic flow steps is very clear to me. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • The sequences of the alternative flow steps are very clear to me. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • The alternative flows clearly correspond to the basic flow. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • It is extremely hard for me to use MS-Visio to draw class diagram. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • It was extremely hard for me to design the class diagram. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Why?

Pre-Lab Questionnaire (Lab 2.2 and 2.4)

Levels of agreement:

1 – Completely agree

2 – Generally agree

3 – Generally disagree

4 – Completely disagree

Questions:

- | | 1 | 2 | 3 | 4 |
|---|--------------------------|--------------------------|--------------------------|--------------------------|
| • I have good knowledge on UML sequence diagram modeling. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I have substantial experience in building sequence diagrams from use cases. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I fully understand how to keep the consistency between class and sequence diagrams. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I am confident I can correctly use MS-Visio to draw sequence diagrams. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Post-Lab Questionnaire (Lab 2.2 and 2.4)

Levels of agreement:

1 – Completely agree

2 – Generally agree

3 – Generally disagree

4 – Completely disagree

Questions:

- | | 1 | 2 | 3 | 4 |
|--|--------------------------|--------------------------|--------------------------|--------------------------|
| • The instructions and objectives of the lab were perfectly clear to me. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I had plenty of time to finish the lab. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • I fully understood the use case descriptions I was provided. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • The sequence of the basic flow steps is very clear to me. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • The sequences of the alternative flow steps are very clear to me. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • The alternative flows clearly correspond to the basic flow. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • It is extremely hard for me use MS-Visio to draw sequence diagrams. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| • It was extremely hard for me to design the sequence diagrams. | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Why?

Appendix M Experiment data

Table 62 Error rates of the restriction rules

Measure	CPD (%)	VS (%)	Avg. (%)	Error rate (%)	Measure	CPD (%)	VS (%)	Avg. (%)	Error rate (%)
R1	1.3	1.93	1.62	2	R17-V	1.1	1	1.05	8
R2	0.87	1	0.94	1	R17-M	28.6	3	15.8	
R3	0.6	0	0.3	0	R18-V	1.4	0.3	0.85	5
R4	0.3	0	0.15	0	R18-M	18	0	9.00	
R5	0	0	0	0	R19-V	0	0	0.00	6
R6	1.5	1.6	1.55	2	R19-M	9.5	12.6	11.05	
R7	13.5	13.9	13.7	14	R20	29.5	12.8	21.15	21
R8	0.1	0.9	0.5	1	R21-V	0	0.2	0.10	6
R9	3.6	9	6.3	6	R21-M	22.2	0	11.10	
R10	0.52	1	0.76	1	R22-M1	10.4	35.6	23.00	25
R11	0.46	1	0.73	1	R22-M2	31.2	22	26.60	
R12	9	6	7.5	8	R23-V	0	0.5	0.25	14
R13	0	0	0	0	R23-M	50	5.6	27.80	
R14	0.72	2	1.36	1	R24-V	3.2	1.6	2.40	3
R15	0.1	0	0.05	0	R24-M	6.7	0	3.35	
R16	0	0	0	0	R25-V	0.8	0	0.40	13
					R25-M	41.7	11	26.35	
					R26	9.9	9.8	9.85	

Table 63 Understandability of the restriction rules

Restriction rule	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
Understandability (%)	90	100	97	90	90	86	90	69	97	65	79	90	69
Restriction rule	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26
Understandability (%)	79	65	100	90	100	90	76	100	100	83	79	83	

Table 64 Applicability of the restriction rules (frequencies)

Level	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
1	0	0	0	0	0	0	4%	0	0	7%	7%	4%	15%
2	0	0	10%	0	3%	15%	4%	21%	7%	22%	14%	0	11%
1+2	0	0	10%	0	3%	15%	8%	21%	7%	29%	21%	4%	26%
3	42%	26%	38%	38%	35%	33%	55%	43%	46%	32%	36%	33%	30%
4	58%	74%	52%	62%	62%	52%	38%	36%	47%	39%	43%	63%	44%
3+4	100%	100%	90%	100%	97%	85%	92%	79%	93%	71%	79%	96%	74%
Level	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26
1	7%	14%	0	0	0	0	0	0	0	0	7%	4%	7%
2	10%	21%	0	0	0	4%	7%	20%	7%	11%	7%	11%	7%
1+2	17%	35%	0	0	0	4%	7%	20%	7%	11%	14%	15%	14%
3	38%	36%	29%	36%	33%	32%	36%	16%	32%	25%	22%	22%	39%

4	45%	29%	71%	64%	67%	64%	57%	64%	61%	64%	63%	64%	47%
3+4	83%	65%	100%	100%	100%	96%	93%	80%	93%	89%	85%	86%	86%

Table 65 Restrictiveness of the restriction rules (frequencies)

Level	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
1	35%	52%	43%	57%	43%	33%	39%	25%	38%	41%	30%	29%	31%
2	43%	37%	34%	27%	37%	49%	44%	47%	47%	38%	38%	46%	34%
1+2	78%	89%	77%	84%	80%	82%	83%	72%	84%	79%	68%	74%	65%
3	19%	11%	14%	17%	17%	12%	14%	28%	16%	21%	32%	20%	29%
4	3%	0	9%	9%	3%	6%	3%	0	0	0	0	6%	6%
3+4	21%	11%	23%	26%	20%	18%	17%	28%	16%	21%	32%	26%	35%
Level	R14	R15	R16	R17	R18	R19	R20	R21	R22	R23	R24	R25	R26
1	29%	31%	47%	34%	35%	40%	31%	36%	31%	32%	34%	36%	32%
2	46%	45%	41%	54%	53%	49%	49%	45%	56%	50%	54%	50%	32%
1+2	74%	76%	88%	88%	88%	88%	80%	82%	86%	82%	88%	86%	64%
3	23%	24%	9%	9%	9%	9%	11%	9%	6%	3%	3%	6%	23%
4	3%	0	3%	3%	3%	3%	9%	9%	8%	15%	9%	8%	13%
3+4	26%	24%	12%	12%	12%	12%	20%	18%	14%	18%	12%	14%	36%

Table 66 Descriptive statistics of measure CD – Lab 2.1 and 2.3

Exp	System	Methods	Class Completeness			Association Completeness			CD Completeness			Class Correctness			CD Correctness			Redundancy		
			Mean	Std Dev	Size	Mean	Std Dev	Size	Mean	Std Dev	Size	Mean	Std Dev	Size	Mean	Std Dev	Size	Mean	Std Dev	Size
1	CPD+VS	UCM_R	0.48	0.17	14	0.11	0.11	14	0.26	0.12	14	0.79	0.13	14	0.88	0.07	14	0.09	0.03	14
		UCM_UR	0.37	0.2	9	0.05	0.07	9	0.18	0.12	9	0.65	0.15	9	0.81	0.07	9	0.14	0.04	9
2.1	CPD	UCM_R	0.66	0.18	8	0.24	0.21	8	0.36	0.06	8	0.76	0.05	8	0.62	0.15	8	0.21	0.07	8
		UCM_UR	0.45	0.12	9	0.08	0.11	9	0.24	0.03	9	0.71	0.08	9	0.64	0.18	9	0.32	0.11	9
	VS	UCM_R	0.64	0.13	9	0.14	0.22	9	0.39	0.15	9	0.86	0.04	9	0.67	0.24	9	0.17	0.08	9
		UCM_UR	0.59	0.07	7	0.04	0.08	7	0.32	0.05	7	0.77	0.11	7	0.73	0.17	7	0.18	0.08	7
	CPD+VS	UCM_R	0.65	0.15	17	0.19	0.22	17	0.38	0.16	17	0.81	0.06	17	0.65	0.2	17	0.19	0.08	17
		UCM_UR	0.51	0.12	16	0.06	0.1	16	0.27	0.09	16	0.76	0.1	16	0.68	0.18	16	0.26	0.12	16
2.3	CPD	UCM_R	0.65	0.1	6	0.21	0.09	6	0.36	0.12	6	0.81	0.03	6	0.51	0.19	6	0.1	0.06	6
		UCM_UR	0.45	0.12	8	0.1	0.11	8	0.23	0.11	8	0.75	0.1	8	0.6	0.16	8	0.2	0.1	8
	VS	UCM_R	0.7	0.14	9	0.1	0.13	9	0.4	0.12	9	0.88	0.06	9	0.83	0.13	9	0.11	0.1	9
		UCM_UR	0.65	0.17	9	0.12	0.12	9	0.39	0.14	9	0.83	0.04	9	0.76	0.17	9	0.16	0.11	9
	CPD+VS	UCM_R	0.68	0.12	15	0.14	0.13	15	0.37	0.11	15	0.85	0.06	15	0.7	0.22	15	0.1	0.08	15
		UCM_UR	0.56	0.18	17	0.11	0.11	17	0.31	0.15	17	0.8	0.08	17	0.68	0.18	17	0.18	0.11	17

Table 67 Descriptive statistics of measure SD – Lab 2.2 and 2.4

Exp	System	Methods	Message Completeness			SD Completeness			Message Correctness			SD Correctness			BCE Consistency		
			Mean	Std Dev	Size	Mean	Std Dev	Size	Mean	Std Dev	Size	Mean	Std Dev	Size	Mean	Std Dev	Size
2.2	CPD	UCM R	0.45	0.05	8	0.67	0.13	8	0.73	0.26	8	0.83	0.14	8	0.95	0.06	8
		UCM UR	0.41	0.13	7	0.61	0.17	7	0.49	0.21	7	0.67	0.18	7	0.91	0.15	7
	VS	UCM R	0.39	0.11	7	0.5	0.18	7	0.72	0.18	7	0.83	0.1	7	1	0.01	7
		UCM UR	0.36	0.12	6	0.51	0.1	6	0.72	0.21	6	0.79	0.13	6	0.95	0.07	6
	CPD+VS	UCM R	0.42	0.09	15	0.59	0.17	15	0.73	0.06	15	0.83	0.12	15	0.97	0.05	15
		UCM UR	0.39	0.12	13	0.56	0.15	13	0.6	0.07	13	0.72	0.17	13	0.93	0.12	13
2.4	CPD	UCM R	0.52	0.16	6	0.77	0.18	6	0.73	0.23	6	0.76	0.17	6	0.9	0.07	6
		UCM UR	0.46	0.1	8	0.61	0.22	8	0.6	0.22	8	0.74	0.17	8	0.9	0.09	8
	VS	UCM R	0.59	0.2	8	0.78	0.14	8	0.84	0.1	8	0.88	0.07	8	0.99	0.03	8
		UCM UR	0.46	0.1	8	0.63	0.1	8	0.61	0.31	8	0.81	0.12	8	0.98	0.03	8
	CPD+VS	UCM R	0.56	0.18	14	0.77	0.15	14	0.79	0.17	14	0.83	0.13	14	0.95	0.11	14
		UCM UR	0.46	0.1	16	0.62	0.16	16	0.61	0.23	16	0.78	0.15	16	0.94	0.18	16

Table 68 ANOVA – Interaction between Method (UCM_R vs. UCM_UR) and System (CPD vs. VS) for CD – Lab 2.1 and Lab 2.3

Exp.	Measures	Factor	DF	Parameter estimate	Sum of Square	F Ratio	Prob > F
2.1	Class Completeness	Method	1	0.0659	0.1419	8.0427	0.0082
		System	1	-0.0299	0.0292	1.6534	0.2087
		Method * System	1	0.0405	0.0535	3.0308	0.0923
	CD Completeness	Method	1	0.05	0.0816	4.8629	0.0355
		System	1	-0.0286	0.0267	1.5893	0.2175
		Method * System	1	0.0119	0.0046	0.2742	0.6045
	Class Correctness	Method	1	0.0368	0.0442	7.877	0.0089
		System	1	-0.0388	0.0491	8.7481	0.0061
		Method * System	1	-0.0079	0.0021	0.3664	0.5497
Redundancy	Method	1	-0.0289	0.0272	3.3946	0.0757	
	System	1	0.0448	0.0655	8.1658	0.0078	
	Method * System	1	-0.0234	0.0179	2.2286	0.1463	
2.3	Class Completeness	Method	1	0.0621	0.1199	6.2663	0.0184
		System	1	-0.0645	0.1297	6.7772	0.0146
		Method * System	1	0.0343	0.0366	1.9125	0.1776
	CD Completeness	Method	1	0.0381	0.0451	3.0662	0.0909
		System	1	-0.0505	0.0794	5.3932	0.0277
		Method * System	1	0.0297	0.0275	1.8709	0.1823
	Class Correctness	Method	1	0.0248	0.0191	4.5373	0.0421
		System	1	-0.0376	0.0441	10.4934	0.0031
		Method * System	1	0.0023	0.0002	0.0405	0.8419
	Redundancy	Method	1	-0.0411	0.0526	5.245	0.0297
		System	1	0.0089	0.0025	0.2475	0.6227
		Method * System	1	-0.0146	0.0066	0.6592	0.4237

Table 69 ANOVA – Interaction between Method (UCM_R vs. UCM_UR) and Order (2.1 vs. 2.3) on CD – Lab 2.1 and Lab 2.3

Measures	Factor	DF	Parameter estimate	Sum of Square	F Ratio	Prob > F
Class Completeness	Method	1	0.0661	0.2833	13.2282	0.0006
	Order	1	-0.0186	0.0225	1.0499	0.3096
	Method * Order	1	0.0039	0.001	0.0465	0.8300
CD Completeness	Method	1	0.0454	0.1336	7.8038	0.007
	Order	1	-0.012	0.0093	0.5415	0.4646
	Method * Order	1	0.0076	0.0038	0.2195	0.6411
Class Correctness	Method	1	0.0336	0.0732	11.83	0.0011
	Order	1	-0.025	0.0406	6.5542	0.013
	Method * Order	1	0.0065	0.0027	0.4432	0.5081
Redundancy	Method	1	-0.0368	0.0878	8.8676	0.0042
	Order	1	0.0412	0.1098	11.0882	0.0015
	Method * Order	1	0.003	0.0006	0.8062	0.8062

Table 70 ANOVA – Interaction between Method (UCM_R vs. UCM_UR) and System (CPD vs. VS) for SD – Experiment 2

Exp.	Measures	Factor	DF	Parameter estimate	Sum of Square	F Ratio	Prob > F
2.2	Message Completeness	Method	1	0.016	0.007	0.6358	0.4331
		System	1	0.027	0.021	1.8732	0.1838
		Method * System	1	0.005	0.001	0.8139	0.8139
	SD Completeness	Method	1	0.012	0.004	0.1682	0.6854
		System	1	0.07	0.137	6.1079	0.0209
		Method * System	1	0.016	0.007	0.3289	0.5717
	Sequence Correctness	Method	1	0.06	0.098	2.0042	0.1689
		System	1	-0.057	0.09	1.8261	0.1892
		Method * System	1	0.059	0.097	1.9723	0.173
	SD Correctness	Method	1	0.051	0.073	3.5595	0.0714
		System	1	-0.027	0.021	1.0218	0.3222
		Method * System	1	0.029	0.023	1.1171	0.3011
BCE Consistency	Method	1	0.022	0.014	1.6849	0.2066	
	System	1	-0.023	0.015	1.8457	0.1869	
	Method * System	1	0.0004	<0.0001	0.0006	0.9802	
2.4	Message Completeness	Method	1	0.047	0.064	2.9904	0.0956
		System	1	-0.017	0.009	0.4007	0.5322
		Method * System	1	-0.015	0.007	0.3176	0.5779
	SD Completeness	Method	1	0.077	0.176	6.5768	0.0165
		System	1	-0.007	0.001	0.0552	0.816
		Method * System	1	0.0004	0.000005	0.0002	0.9891
	Message Correctness	Method	1	0.089	0.234	4.5339	0.0429
		System	1	-0.03	0.026	0.5038	0.4842
		Method * System	1	-0.025	0.019	0.3662	0.5503
	SD Correctness	Method	1	0.021	0.012	0.6667	0.4216
		System	1	-0.048	0.067	3.5879	0.0694
		Method * System	1	-0.013	0.005	0.2515	0.6202
	BCE Consistency	Method	1	0.004	0.001	0.0207	0.8866
		System	1	-0.051	0.051	2.2964	0.1417
		Method * System	1	<0.0001	<0.0001	0.0001	0.9904

Table 71 ANOVA – Interaction between Method (UCM_R vs. UCM_UR) and Order (2.2 vs. 2.4) on SD – Lab 2.2 and Lab 2.4

Measures	Factor	DF	Parameter estimate	Sum of Square	F Ratio	Prob > F
Message Completeness	Method	1	0.025	0.037	0.8815	0.352
	Order	1	-0.222	2.83	67.5539	<.0001
	Method * Order	1	-0.009	0.005	0.1153	0.7355
SD Completeness	Method	1	0.053	0.16	4.1777	0.0458
	Order	1	-0.062	0.223	5.8071	0.0194
	Method * Order	1	-0.04	0.093	2.4363	0.1244
Message Correctness	Method	1	0.044	0.114	3.2296	0.0779
	Order	1	-0.07	0.282	7.9774	0.0066
	Method * Order	1	0.02	0.022	0.6302	0.4308
SD Correctness	Method	1	0.03	0.052	2.4339	0.1246
	Order	1	-0.083	0.396	18.4816	<.0001
	Method * Order	1	0.023	0.031	1.4446	0.2346
BCE Consistency	Method	1	0.022	0.028	2.0863	0.1544
	Order	1	0.024	0.033	2.4192	0.1257
	Method * Order	1	0.0003	<0.0001	0.0005	0.983

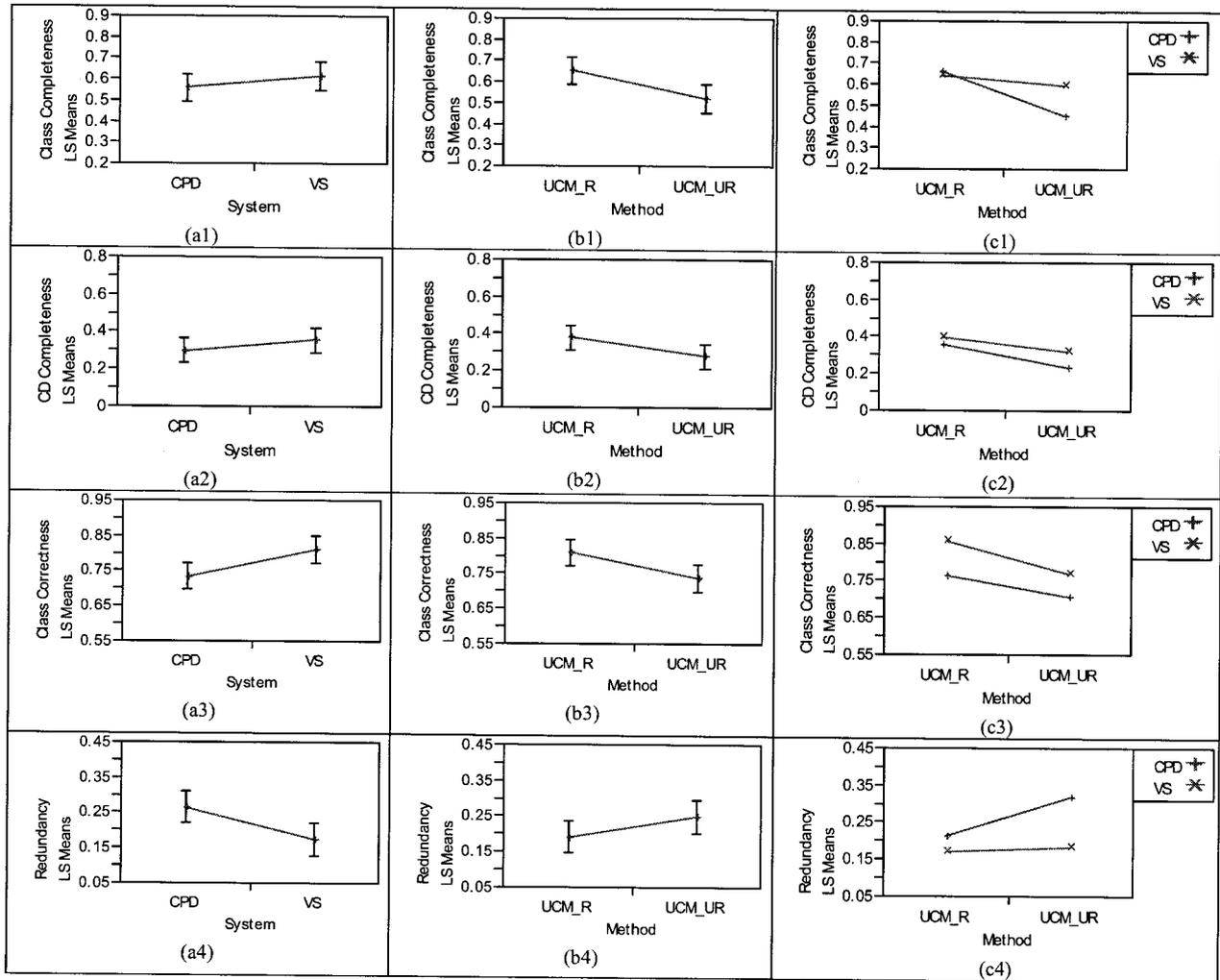


Figure 113 Least-square means for ANOVA interaction analysis between Method and System for CD - Lab 2.1

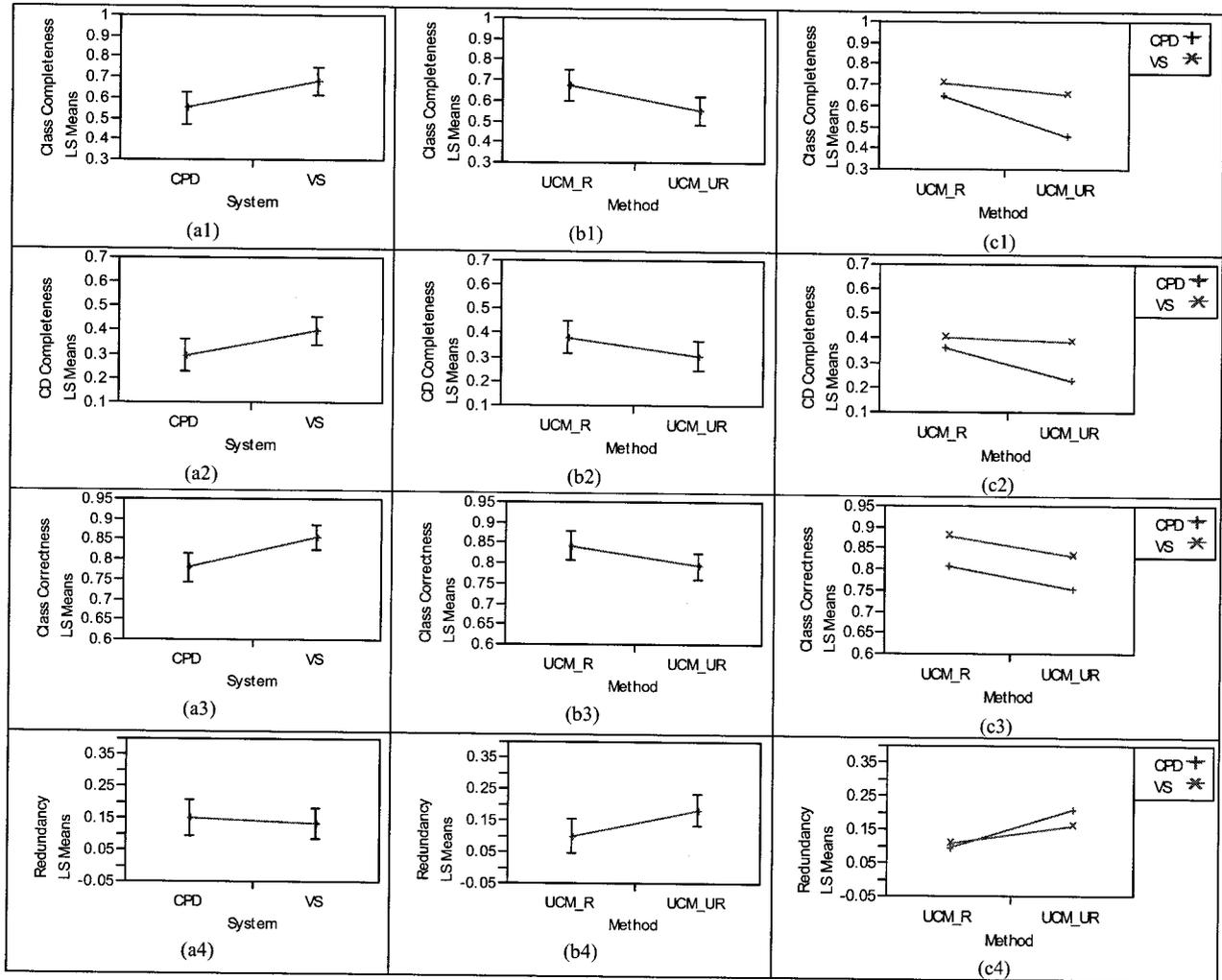


Figure 114 Least-square means for ANOVA interaction analysis between Method and System for CD - Lab 2.3

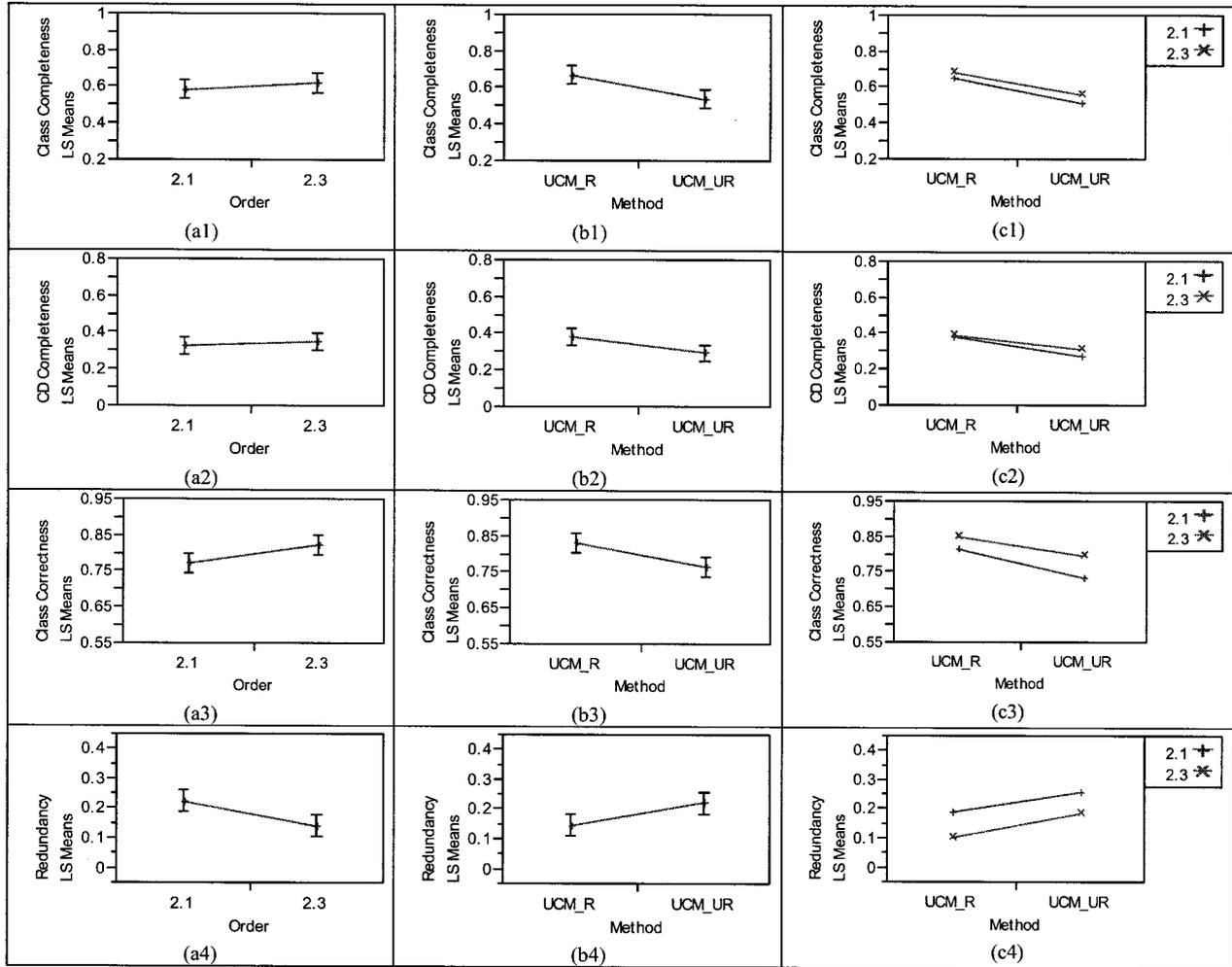


Figure 115 Least-square means for ANOVA interaction analysis between Method and Order for CD - Lab 2.1 and Lab 2.3

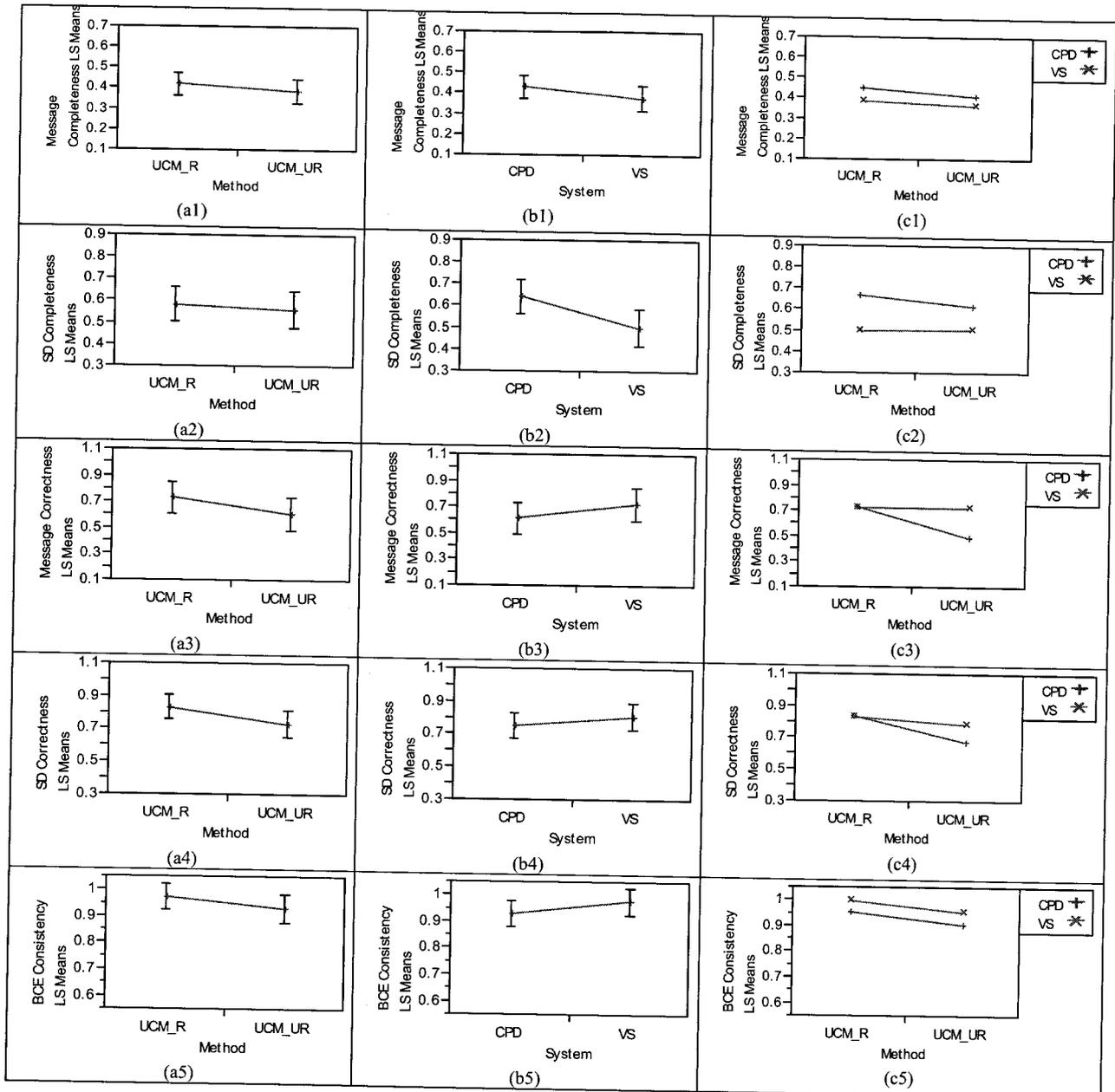


Figure 116 Least-square means for ANOVA interaction analysis between Method and System for SD - Lab 2

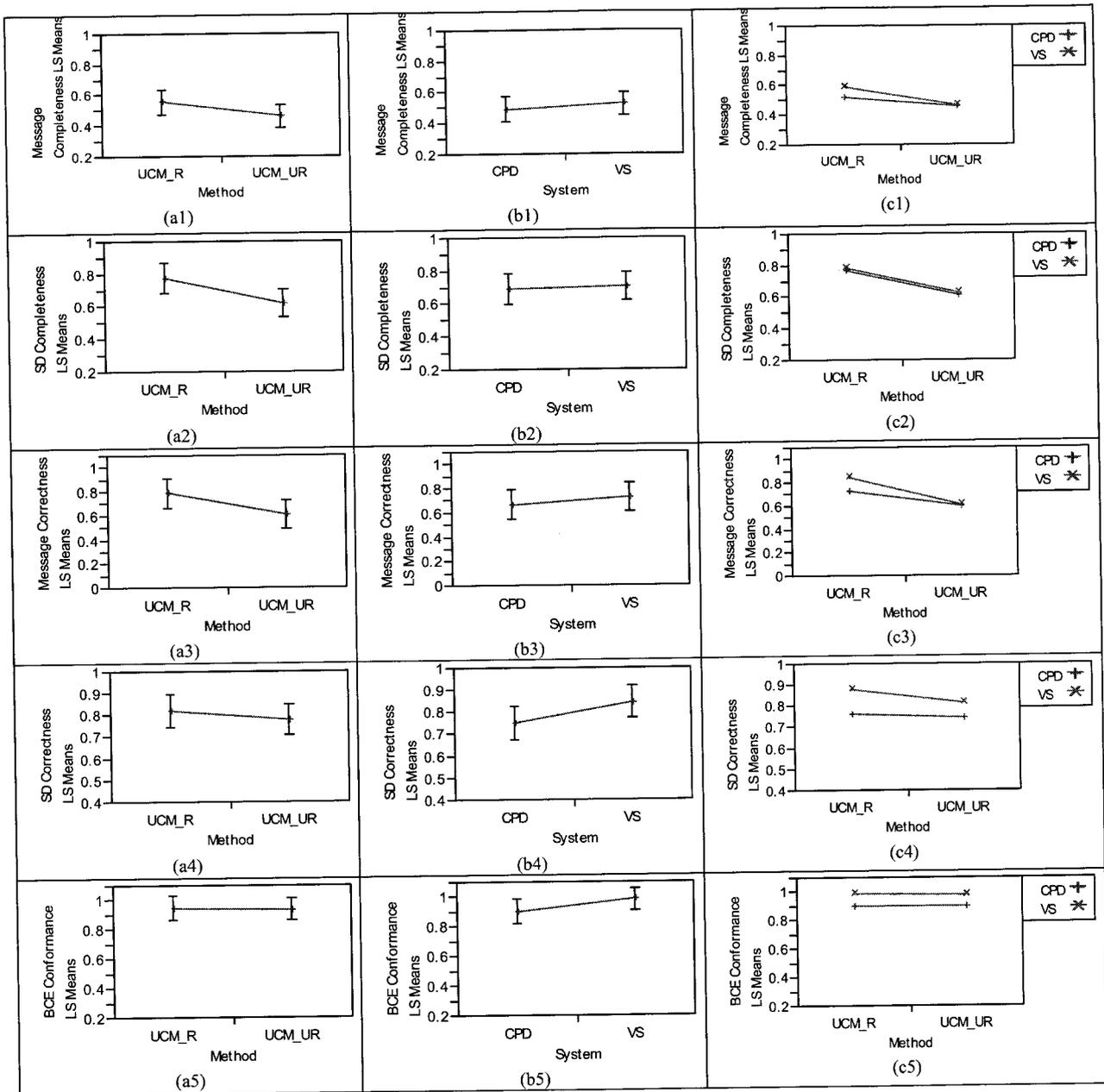


Figure 117 Least-square means for ANOVA interaction analysis between Method and System for SD - Lab 2.4

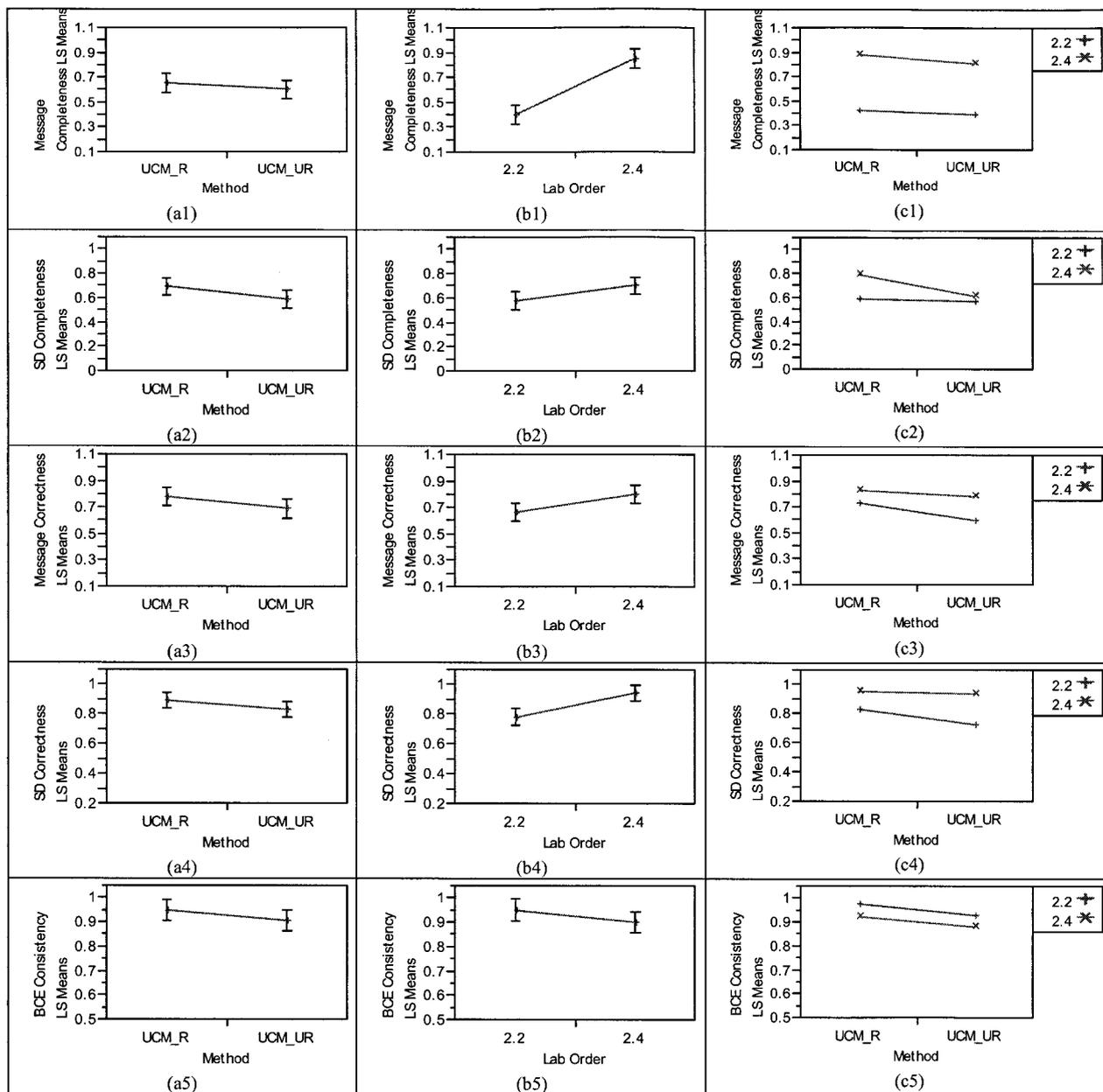


Figure 118 Least-square means for ANOVA interaction analysis between Method and Order for SD - Lab 2.2 and Lab 2.4