

**EMBRACING MODEL TRANSFORMATIONS IN FUNCTIONAL
REQUIREMENTS SPECIFICATION**

BY

YASSER ALI KHAN

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In
SOFTWARE ENGINEERING

MAY 2013

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN- 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **YASSER ALI KHAN** under the direction of his thesis advisor and approved by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN SOFTWARE ENGINEERING**.



Dr. Mohamed El-Attar
(Advisor)



Dr. Adel Ahmed
Department Chairman



Dr. Mahmoud Elish
(Member)



Dr. Salam A. Zummo
Dean of Graduate Studies



Dr. Sajjad Mahmood
(Member)

20/5/13

Date

© Yasser Ali Khan

2013

Dedication

This thesis is dedicated in the loving memory of my late father Bahbood Ali Khan. He witnessed the inception of this work, but left this world before its completion. His aspiration that I pursue graduate studies motivated me to continue this work in the darkest of times. Words cannot express the efforts he constantly made towards my education. May Allah (SWT) be pleased with him and grant him highest place in Jannah (Ameen). The constant prayers of my mother Khalida Khatoon have enabled me to complete this strenuous journey. I thank my sisters Nadia Khatoon, and Hania Khatoon, for taking responsibility while I have been away these years. I highly appreciate the support given by my brother-in-law Afzaal Ahmed Khan towards my career. I dearly miss my three lovely nieces Raazia Fatima Khatoon, Marzia Fatima Khatoon, and Zaakia Fatima Khatoon. I love you all, and can't wait to be home.

ACKNOWLEDGMENTS

First of all I would like to thank almighty Allah (Subhanahu Wa Ta'ala) for giving me the ability to accomplish this thesis. Peace and blessing of Allah (Subhanahu Wa Ta'ala) be upon his last messenger Mohammed (Sallallahu Alaihi Wasallam), who guided us on the right path.

I would like to convey my sincere gratitude to my advisor Dr. Mohamed El-Attar for his guidance and encouragement he provided throughout this research. I feel very fortunate to have had the opportunity to work under the supervision of Dr. Attar. I fondly remember the brief chat we had back in November 2011 in which Dr. Attar expressed the idea for this thesis, and since then there has been no turning back. His immaculate vision paved the path for the smooth, timely, and successful completion of this thesis. I extremely appreciate his invaluable advice towards my future academic endeavors. I consider Dr. Attar not only as an outstanding supervisor, but also as my elder brother.

I would also like to thank the committee members, Dr. Mahmoud Elish and Dr. Sajjad Mahmood, for their feedback towards this thesis. A number of people that I would like to thank for taking time out of their busy schedules to review the initial work on Chapter 4, namely Dr. Mahmoud Elish, Dr. Mohammad Alshayeb and Dr. Jameleddine Hassine. The remarks from Dr. Sajjad Mahmood towards the initial work on Chapter 5 are also appreciated. Last but not least, I thank Dr. Jameleddine Hassine for his insightful comments and reviews on Chapter 6.

TABLE OF CONTENTS

ACKNOWLEDGMENTS	V
TABLE OF CONTENTS	VI
LIST OF TABLES	XII
LIST OF FIGURES	XIV
LIST OF CODE LISTINGS	XVIII
LIST OF ABBREVIATIONS	XX
ABSTRACT (ENGLISH).....	XXIII
ABSTRACT (ARABIC)	XXV
CHAPTER 1 INTRODUCTION.....	1
1.1 Model Transformation	5
1.2 Antipatterns.....	7
1.3 UCM Notation.....	8
1.4 UML 2 AD Notation	10
1.5 UML 2 SD Notation	11
1.6 Atlas Transformation Language	13
1.6.1 ATL Rules.....	14
1.6.2 ATL Helpers.....	16

1.6.3	ATL Execution Modes	16
1.6.4	ATL Module Superimposition	17
1.7	Research Question	19
1.8	Thesis Objectives	20
1.9	Research Methodology	20
1.10	Thesis Outline	23
 CHAPTER 2 LITERATURE REVIEW.....		24
2.1	Use Case Quality Improvement.....	24
2.1.1	Using Inspection, Guidelines, and Templates	24
2.1.2	Using Automated Verification Tools	26
2.1.3	Use Case Refactoring	26
2.2	Antipatterns.....	28
2.2.1	Impact of Antipatterns on Quality Attributes	28
2.2.2	Antipattern Detection	28
2.3	Model Transformation based Model Refactoring.....	30
2.4	Verification of Model Transformations	31
2.5	UCM transformations	34
 CHAPTER 3 A MODEL TRANSFORMATION APPROACH TOWARDS REFACTORING USE CASE MODELS BASED ON ANTIPATTERNS		36
3.1	Use Case Modeling Antipattern Refactorings	39
a1.	Accessing a <i>generalized concrete</i> use case	39

a2.	Accessing an <i>extension</i> use case	43
a3.	Using <i>extension/inclusion</i> use cases to implement an <i>abstract</i> use case	48
a4.	Functional Decomposition: Using the <i>include</i> relationship	53
a5.	Functional Decomposition: Using the <i>extend</i> relationship.....	57
a6.	Multiple <i>generalizations</i> of a use case	62
a7.	Use cases containing common and exceptional functionality	64
a8.	Multiple actors associated with one use case	68
a9.	An association between two actors	71
a10.	An association between use cases	72
a11.	An unassociated use case	74
a12.	Two actors with same name	75
a13.	An actor associated with an unimplemented <i>abstract</i> use case	77
3.2	Case Study	80
3.2.1	Definition and Motivation	81
3.2.2	Formulation	81
3.2.3	Model Transformations	85
3.3	Evaluation.....	94
CHAPTER 4 AUTOMATED TRANSFORMATION OF USE CASE MAPS TO UML 2 ACTIVITY DIAGRAMS		96
4.1	UCM to UML 2 AD mappings.....	97
4.2	Transformation Rules.....	101
4.2.1	Entry point and Matched Rule.....	101

4.2.2	Lazy Rules.....	102
4.2.3	Called Rules	104
4.2.4	Helpers	106
4.3	Case Studies.....	106
4.3.1	Elevator Control System	106
4.3.2	Mock System.....	109
4.4	Target Model Verification.....	113
CHAPTER 5 DERIVING UML 2 SEQUENCE DIAGRAMS FROM USE CASE MAP SCENARIO SPECIFICATIONS		115
5.1	UCM to UML 2 SD mappings.....	116
5.1.1	Components and Responsibilities	116
5.1.2	OR-forks.....	120
5.1.3	AND-forks.....	127
5.1.4	Waiting Point	128
5.1.5	Timer	130
5.1.6	Failure Point	132
5.1.7	Nested Components.....	133
5.1.8	Stub	134
5.1.9	Dynamic Stubs	136
5.2	Transformation Rules.....	137
5.3	Case Study	140
5.3.1	Source Model	140

5.3.2	Scenario Extraction	143
5.3.3	Transformation	145
CHAPTER 6 A MUTATION FRAMEWORK FOR MODEL TRANSFORMATIONS		154
6.1	ATL Mutation Testing Approach.....	155
6.2	ATL Mutation Operators.....	157
6.2.1	Matched to Lazy (M2L)	158
6.2.2	Lazy to Matched (L2M)	158
6.2.3	Delete Attribute Mapping (DAM).....	159
6.2.4	Add Attribute Mapping (AAM)	160
6.2.5	Delete Filtering Expression (DFE).....	161
6.2.6	Add Filtering Expression (AFE)	162
6.2.7	Change Source Type (CST)	163
6.2.8	Change Target Type (CTT).....	164
6.2.9	Change Execution Mode (CEM).....	165
6.2.10	Delete Return Statement (DRS)	166
6.3	ATL Mutation Operator Analysis.....	166
6.3.1	Number of Generated ATL Mutants	168
6.3.2	Equivalent ATL Mutants.....	169
6.3.3	Other Remarks	170
6.4	MuATL (Mutation Toolkit for ATL)	171
6.5	Case Study: UCM to UML 2 AD Transformation.....	172

6.5.1	Test Cases.....	173
6.5.2	Generated Mutants	175
6.5.3	Mutation Analysis Results.....	176
6.5.4	Test Suite Enhancement	177
6.6	Discussion	180
CHAPTER 7 CONCLUSION AND FUTURE WORK.....		182
7.1	Thesis Summary.....	182
7.2	Future Work.....	185
REFERENCES.....		187
APPENDIX A – UCM METAMODEL		203
APPENDIX B – UML 2 AD METAMODEL		204
VITAE.....		205

LIST OF TABLES

Table 1: Example of matched, lazy, mapping called, non-mapping called rules	16
Table 2: Example of ATL helper and attribute	16
Table 3: Example of modules in default mode and refining mode	17
Table 4: Use case antipatterns and their respective refactorings	37
Table 5: Antipatterns matched in the use case models of MAPSTEDI, and the refactorings applied.....	94
Table 6: All possible scenarios in ECS UCM.....	143
Table 7: Example of a M2L mutation.....	158
Table 8: Example of a L2M mutation.....	159
Table 9: Example of a DAM mutation	160
Table 10: Example of an AAM mutation	161
Table 11: Example of a DFE mutation	162
Table 12: Example of a AFE mutation	163
Table 13: Example of a CST mutation	164
Table 14: Example of a CTT mutation	164
Table 15: Example of a CEM mutation	165
Table 16: Example of a DRS mutation.....	166
Table 17: Summary of ATL mutation operators.....	167

Table 18: Test cases of UCM to UML AD model transformation 173

Table 19: CST and CTT mutant corresponding to lazy rule
Responsibility_To_OpaqueAction..... 176

Table 20: Types of mutants created for the UCM to UML 2 AD model transformation 177

LIST OF FIGURES

Figure 1: UCM notation.....	9
Figure 2: UML 2 AD notation	11
Figure 3: UML2 SD notation.....	13
Figure 4: Overview of ATL model transformations.....	14
Figure 5: Example of Concrete to Abstract refactoring.....	41
Figure 6: Example of Drop Generalized UC Association refactoring.....	43
Figure 7: Example of Drop Actor-Extension UC Association refactoring.....	45
Figure 8: Example of Directed Actor-Extension UC Association refactoring	47
Figure 9: Example of Abstract Extended UC to Concrete refactoring	50
Figure 10: Example of Inclusion to Generalization refactoring	52
Figure 11: Example of Drop Functional Decomposition refactoring	55
Figure 12: Example of Drop Functional Decomposition having Inclusion refactoring ...	57
Figure 13: Example of Spilt Extension UC refactoring.....	60
Figure 14: Example of Extension to Generalization refactoring	62
Figure 15: Example of Generalization to Inclusion refactoring	64
Figure 16: Example of Drop Inclusion and Drop Extension refactorings	67
Figure 17: Example of Generalize Actors refactoring	70

Figure 18: Example of Drop UC-UC Association refactoring	74
Figure 19: Example of Rename Actor refactoring.....	77
Figure 20: Example of Abstract to Concrete refactoring.....	79
Figure 21: Example of Add Concrete UC refactoring.....	80
Figure 22: Use case model of Database Access subsystem.....	83
Figure 23: Use case model of Database Queries subsystem.....	83
Figure 24: Use case model of Database Integrator subsystem	84
Figure 25: Use case model of Database Edits subsystem.....	84
Figure 26: Use case model of Administrative Process subsystem.....	85
Figure 27: Use case model of Database Access subsystem after applying the Generalize Actors refactoring	86
Figure 28: Use case model of Database Queries subsystem after applying the Extension to Generalization Refactoring	87
Figure 29: Use case model of Database Integrator subsystem after applying the Drop Functional Decomposition refactoring	88
Figure 30: Use case model of Database Integrator subsystem after applying the Drop Functional Decomposition with Include refactoring on the use case model in Figure 29	89
Figure 31: Use case model of Database Edits subsystem after applying the Drop Functional Decomposition refactoring	90
Figure 32: Use case model of Administrative Process subsystem after applying the Split UCs refactoring.....	91

Figure 33: Merged use case model of Database Queries and Database Integrator subsystem.....	92
Figure 34: Use case model of Database Queries and Database Integrator subsystem after applying the Concrete to Abstract refactoring.....	93
Figure 35: Mapping of UCM to UML 2 AD notation	100
Figure 36: Elevator Control System source UCM.....	108
Figure 37: Elevator Control System target AD.....	109
Figure 38: Mock System source UCM	111
Figure 39: Mock System Stub NC	111
Figure 40: Mock System target AD	113
Figure 41: Mapping of components and responsibilities.....	118
Figure 42: Mapping of bounded start and end points	120
Figure 43: Mapping of alternate paths	122
Figure 44: Mapping of terminating alternate path	123
Figure 45: Mapping of a UCM loop	125
Figure 46: Alternate mapping of the UCM loop shown in Figure 45.....	126
Figure 47: Mapping of concurrent paths.....	128
Figure 48: Mapping of waiting points	130
Figure 49: Mapping of timers	131
Figure 50: Mapping of failure points	133

Figure 51: Mapping of nested components.....	134
Figure 52: Mapping of stubs.....	135
Figure 53: Mapping of dynamic stubs	137
Figure 54: Elevator Control System UCM	142
Figure 55: Scenarios S1, S5, S8 and S12 of the Elevator Control System UCM.....	145
Figure 56: Mapping of scenario S1 to SD notation	146
Figure 57: Mapping of scenario S5 to SD notation	146
Figure 58: Mapping of scenario S8 to SD notation	149
Figure 59: Mapping of scenario S12 to SD notation	153
Figure 60: ATL mutation process	156
Figure 61: MuATL GUI.....	171
Figure 62: AFE Mutant GUI.....	172
Figure 63: Input and expected output models of TC1	174
Figure 64: Input and expected output models of TC2	174
Figure 65: Input and expected output models of TC8	178
Figure 66: Input and expected output models of TC9	178
Figure 67: Input and expected output models of TC10	180
Figure 68: UCM metamodel	203
Figure 69: UML 2 AD metamodel.....	204

LIST OF CODE LISTINGS

Listing 1: ATL rule for applying Concrete to Abstract refactoring.....	40
Listing 2: ATL rule for applying Drop Actor-Generalized UC Association refactoring .	42
Listing 3: ATL rule for applying Drop Actor-Extension UC Association refactoring.....	44
Listing 4: ATL rules for applying Directed Actor-Extension UC Association refactoring	46
Listing 5: ATL rule for applying Abstract Extended UC to Concrete refactoring	49
Listing 6: ATL rule for applying Inclusion to Generalization refactoring	51
Listing 7: ATL rule for applying Drop Functional Decomposition refactoring	54
Listing 8: ATL for applying Split Extension UC refactoring	59
Listing 9: ATL rule for applying Extension to Generalization refactoring	61
Listing 10: ATL rule for applying Generalization to Inclusion refactoring	63
Listing 11: ATL rule for applying Drop Inclusion refactoring.....	65
Listing 12: ATL rule for applying Drop Extension refactoring.....	66
Listing 13: ATL rule for applying Generalize Actor refactoring	69
Listing 14: ATL rule for applying Split UCs refactoring	71
Listing 15: ATL rule for applying Drop Actor-Actor Association refactoring	72
Listing 16: ATL rule for applying Drop UC-UC Association refactoring	73
Listing 17: ATL rule for applying Drop Unassociated UC refactoring.....	75

Listing 18: ATL rule for applying Rename Actor refactoring.....	76
Listing 19: ATL rule for applying Abstract to Concrete refactoring.....	78
Listing 20: ATL rule for applying Add Concrete UC refactoring	80
Listing 21: The entry point rule	102
Listing 22: The matched rule	102
Listing 23: Lazy rules	103
Listing 24: Called rules	105
Listing 25: Helper rules	106
Listing 26: UCM to UML 2 SD ATL mapping rules	140

LIST OF ABBREVIATIONS

AAM	:	Add Attribute Mapping
AD	:	Activity Diagram
AFE	:	Add Filtering Expression
ARBIUM	:	Automated Risk-Based Inspection of Use-Case Models
ASCC	:	All Source Classes Criteria
ATL	:	Atlas Transformation Language
CACA	:	Classes' association creation addition
CACD	:	Classes' association creation deletion
CFCA	:	Collection filtering change with addition
CFCD	:	Collection filtering change with deletion
CEM	:	Change Execution Mode
CST	:	Change Source Type
CTT	:	Change Target Type
DAM	:	Delete Attribute Mapping
DBG	:	Denver Botanic Gardens

DFE	:	Delete Filtering Expression
DMNS	:	Denver Museum of Nature and Science
DRS	:	Delete Return Statement
EC	:	Elevator Control System
ECS	:	Elevator Control System
EM	:	Elevator Manager
EMF	:	Eclipse Modeling Framework
FRS	:	Functional Requirements Specification
GUI	:	Graphical User Interface
KAOS	:	Knowledge Acquisition in Automated Specification
L2M	:	Lazy to Matched
MAPSTEDI	:	Mountains and Plains Spatio-Temporal Database Informatics
MDE	:	Model Driven Engineering
MSC	:	Message Sequence Chart
MuATL	:	Mutation Toolkit for Atlas Transformation Language
M2L	:	Matched to Lazy

OCL	:	Object Constraint Language
OMG	:	Object Management Group
OO	:	Object-Oriented
QVT-O	:	Query View Transformation-Operational
QVT-R	:	Query View Transformation-Relational
ROOM	:	Real-Time Object-Oriented Modeling
SD	:	Sequence Diagram
SP	:	Status and Planner
TC	:	Test Case
TS_{eff}	:	Test Set effectiveness
UC	:	Use Case
UCM	:	Use Case Maps
UCOM	:	University of Colorado Museum
UML	:	Unified Modeling Language
URN	:	User Requirements Notation

ABSTRACT (ENGLISH)

Full Name : Yasser Ali Khan
Thesis Title : Embracing Model Transformations in Functional Requirements Specification
Major Field : Software Engineering
Date of Degree : May 2013

Functional Requirements Specification (FRS) is a software process activity that involves documenting the intended behavior of a system-to-be. Use case modeling is a common approach used in FRS for Object-Oriented systems. Since use case modeling is performed early in a software development cycle, any defects in a use case model will propagate to subsequent development phases and artifacts. Therefore, it is crucial to produce high quality use case models, especially in use case-driven approaches. Previous work on use case quality improvement performed manual refactoring on use case models. Use case models of large scale complex software systems usually contain thousands of use cases. For such use case models, manual refactoring will be prone to human errors, leading to new defects being injected into the models. In order to avoid this issue, a fully automated process for carrying out the refactorings is necessary. Another approach used in FRS is scenario modeling, which is performed in conjunction with use case modeling. Uses cases are described in natural language as scenarios, which are modeled in detail as UML activity and sequence diagrams. A large conceptual gap exists between use cases

and UML design; consequently, developers may produce UML models that do not accurately represent the required behavior of a use case. The Use Case Map (UCM) scenario modeling notation aids in bridging this conceptual gap. However, to date, the UCM notation is not part of the UML modeling language. As such, there lacks research in the area of transforming UCMs into UML design models. Model transformation is an automated technique that can greatly improve several software development activities. This thesis presents an approach that leverages model transformation to execute use case model refactorings, and transform UCM scenario specifications into UML 2 activity diagram and sequence diagram notations. The proposed approach will present a case for software developers to embrace the notion of model transformation in the context of FRS. Furthermore, a fault-based technique is proposed for thorough verification of model transformations. Case studies are presented for evaluating the effectiveness of the proposed approach. The results obtained show that model transformations can efficiently improve FRS by saving time and effort.

ABSTRACT (ARABIC)

ملخص الرسالة

الاسم الكامل: ياسر علي خان

عنوان الرسالة: الاستفادة من تقننة "تحويل نماذج البرامج" لاعدة تنشيط صناعة أنظمه البرمجيات

التخصص: هندسة البرمجيات

تاريخ الدرجة العلمية: مايو ٢٠١٣

توصيف متطلبان النظام هي عبارته عن عملية تتضمن توثيق الوظائف المطلوبة في النظام، نموذج حالة الاستخدام عبارته عن تقنيته او طريقته مشتركه تستخدم في توصيف متطلبات النظم غرضية التوجه. بما إن نموذج حالة الاستخدام يتم تنفيذها في وقت مبكر من دورة تطوير البرمجيات ; فإن أي عيوب في نموذج حالة الاستخدام يتم نشرها إلى مراحل التطور اللاحق والوثائق. لذا فمن الأهمية بمكان أن يتم إنتاج نماذج حالة استخدام ذات جودة عالية، وخاصة في استخدام طرق ممنهجه لحالات الاستخدام. العمل السابق بشأن تحسين جودة حالة الاستخدام بإجراء إعادة صياغته لنماذج حالة الاستخدام. نماذج حالة الاستخدام في النظم المعقدة في العادة تتكون من الآلاف من حالات الاستخدام. على سبيل المثال حالة الاستخدام وإعادة الصياغ لحالة الاستخدام بشكل يدوي تكون عرضه للأخطاء البشرية , مما يؤدي إلى عيوب جديدة التي يجري حقنها في نماذج من أجل تجنب هذه المشكلة يتم عملية إعادة الصياغ لحالات الاستخدام بطريقة مؤتمتة بالكامل. وهناك طريقة أخرى في توصيف متطلبات النظام هو النمذجة باستخدام السيناريو، والتي تتم بالتزامن مع استخدام النمذجة لحالة الاستخدام. حالات الاستخدام يتم وصفها باللغة الطبيعية كسيناريوهات , والتي تتمودج بتفصيل باستخدام لغة النمذجة الموحدة والإشكال المتسلسله. توجد هناك فجوه كبيره بين حالات الاستخدام وتصاميم لغة النمذجة الموحدة وبالتالي فإن المصممون ربما يقومون بإنتاج نماذج من لغة النمذجة الموحدة والتي لا تمثل بدقة السلوك المطلوب لحالات الاستخدام. خرائط حالات الاستخدام تعمل على سد هذه الفجوة. ومع ذلك، حتى الآن فإن رموز حالة الاستخدام لا تعتبر جزء من لغة النمذجة الموحدة.

على هذا النحو فإن عملية التحويل من حالات الإستخدام الى تصاميم لغة النمذجة الموحدة تقتقر لكثير من البحوث. نموذج التحويل هو عبارته عن أسلوب آلي يمكن عن طريقه تحسين الكثير من أنشطة تطوير البرمجيات. يقدم هذا البحث أسلوب او طريقته تمكن نموذج التحويل من تنفيذ إعادة الصياغ لحالات الاستخدام و تحويل توصيفات السيناريو لحالات الاستخدام إلى مخطط الأنشطة للغة النمذجة الموحدة الثاني والإشكال المتسلسلة. والنهج المقترح تقديم الحال بالنسبة لمطوري البرمجيات لتبني مفهوم التحول النموذجي في سياق وظيفية مواصفات المتطلبات . وعلاوة على ذلك، تم إقتراح تقنية على أساس الخطأ للتحقق من التحولات النماذج. تم تقييم فاعلية هذا التكنيك باستخدام بعض التجارب او دراسات الحالة. النتائج التي تم الحصول عليها تثبت بان هذا النموذج لتحويل يساعد في تطوير توصيفات متطلبات النظم عن طريق حفظ الوقت والجهد

CHAPTER 1

INTRODUCTION

Functional Requirements Specification (FRS) is a software process activity that involves documenting the intended behavior of a system-to-be. Use case modeling is a common approach used in FRS for Object-Oriented (OO) systems. A UML [159] use case model provides a visual summary of the use cases, actors and their relationships. Use cases are descriptions of services provided by the system, and actors represent the entities that require these services. Since use case modeling is performed early in a software development cycle, any defects in a use case model will propagate to subsequent development phases and artifacts. The cost of fixing defects in later phases is three to six times more than during requirements engineering [154]. Moreover, requirements defects are most common reason for project failure, and budget overruns [151]. Therefore, it is crucial to produce high quality use case models, especially in use case-driven approaches. To this end, early detection of defects in use case models will significantly improve overall product quality.

Another approach used in FRS is scenario modeling, which is performed in conjunction with use case modeling. Uses cases are described in natural language as scenarios, which are modeled in detail as UML activity and sequence diagrams. A *scenario* is a sequence of interactions, including invariants, between actor and system that

are performed in order to yield an observable result to the actor. A large conceptual gap exists between use cases and UML design [10]; consequently, developers may produce UML models that do not accurately represent the required behavior of a use case.

Use case modeling guidelines [111][157], use case description templates [32][45], and automated use case verification tools [29][160] have been proposed in the literature as means for improving the quality of use case models. In addition to these approaches, the concept of source code refactoring has been extended for use case models in order to improve their quality [40][158][188][191]. In earlier work, El-Attar et al. [18][19][20][22] presented an antipatterns based approach to improve quality in use case models. An antipattern based approach is one that is based on learning from previous experiences and mistakes. Antipatterns are textually described to help its users understand, detect and fix designs that are likely to have harmful consequences downstream. In order to fix problematic designs, an antipattern usually prescribes a set of refactorings to be applied to the use case model. In their technique, antipattern detection is performed in a semi-automated manner. However, the required refactoring tasks are carried out manually. Use case models of large scale complex software systems usually contain thousands of use cases [29]. For such use case models, performing the prescribed refactorings manually will be prone to human errors, leading to new defects being injected into the models. In order to avoid this issue, a fully automated process for carrying out the refactorings is necessary.

The Use Case Map (UCM) [38][39] scenario modeling notation aids in bridging the conceptual gap between natural language scenarios and high-level design. UCMs have

been successfully used for documenting scenarios in telecommunication systems [9][14], web applications [6], agent based systems [2][59], and operating systems [31]. Moreover, UCM is a competitive modeling language, and offers additional benefits compared to other notations [139]. These benefits include integration with goal models in the URN; support for modularization of complex scenarios; integration with simple a metamodel, performance annotations, and a simple action language for analysis. Amyot et al. [10] proposed an extension of UML 1.3 with UCM core concepts for the purpose of introducing a new “UCM View” to the existing set of UML views. However, to date, the proposed “UCM View” is not a UML standard. As such, there lacks research in the area of transforming UCMs into UML design models.

Model transformation is an automated technique that can greatly improve several software development activities including model refactoring [169]. Model transformations approaches have been proposed in the literature for applying design patterns [190]; refactoring UML class diagrams [119][153], UML activity diagrams [58], and KAOS models [42]; and product line evolution [150]. This thesis presents an approach that leverages model transformation to execute use case model refactorings, and transform UCM scenario specifications into UML 2 Activity Diagram (AD) and Sequence Diagram (SD) notations.

Faults in model transformations may result in defective models, and eventually defective code. Correction of defects at the code level is considered very late and is often expensive. Hence, defects must be detected and rectified early in the software process. Uncorrected defects in the models will propagate to other artifacts; thus, adversely affect

the quality of the end product. Moreover, defect propagation may result in a system that does not meet the stakeholders' requirements. Therefore, model transformations must be thoroughly tested to maintain product quality while keeping development cost at reasonable levels. Although, verification of model transformations may benefit from existing software testing techniques, the nature of the input and output data manipulated by transformations makes these activities more complex. Indeed, transformation programs manipulate models, which are complex data structures, making the problem of test data generation and selection, as well as oracle definitions, very difficult [15]. In the literature, many model transformation testing approaches have been studied. These approaches range from partial to full validation of the transformation's behavior and associated properties.

The mutation testing technique is considered as the "gold standard" of software testing. Several studies in the literature have empirically evaluated the effectiveness of mutation testing on traditional programs [84]. It has been shown that mutation testing detects more faults than coverage based techniques. Existing literature on testing model transformations has considered these techniques. Therefore, there is a need to perform mutation testing of model transformations. In order to do so, mutation operators must be defined for the various model transformation languages. Previous work [136] on mutation testing of model transformations defined generic mutation operators that must be adapted for different model transformation languages. In this thesis, a suite of mutation operators are proposed for the Atlas Transformation Language (ATL) [85][175], so that model transformation developers can practice mutation testing; therefore, gain its benefits.

1.1 Model Transformation

Model Driven Engineering (MDE) [89][162][172] is an approach to software development that allows developers to focus on high-level abstractions (*models*) of software rather than low-level implementations (code). In MDE, models can be refined to lower levels of abstraction, refactored to improve maintainability and readability, transformed to other models, and used to generate code [169]. The MDE approach aims to provide automated support to carry out these tasks.

One of the key components of the MDE approach is *model transformation*. A model transformation is the automated translation of a *source* model to a *target* model based on a set of transformation *rules* [96]. A rule defines how elements in a source model map to elements in a target model. The source and target models must conform to a well defined *metamodel*, which specifies the language (syntax and semantics) of the models [64].

Model transformations can be categorized in a number of ways [128]. Based on the number of source and target models there are *one-to-one*, *one-to-many*, *many-to-one* and *many-to-many* model transformations. If the source and target models conform to the same metamodel, their model transformation is referred as *endogenous*. The model transformations presented in Section 3.1 are endogenous. *Exogenous* model transformations are transformations between models which conform to distinct metamodels. For example, a model transformation that derives a UML class diagram

from multiple UML SDs is a many-to-one exogenous model transformation. The UCM to AD, and UCM to SD model transformations presented in Section 4.2 and Section 5.2, respectively, are exogenous. Endogenous model transformations are further classified into *in-place* and *out-place*. In an in-place transformation, a single model serves as both source and target; whereas in an out-place transformation, the source is read only, while the target model is write only. Exogenous transformations are always *out-place* since the source and target models are of distinct type. A *vertical* model transformation results in the source and target models at different levels of abstraction, whereas in a *horizontal* model transformation, they are at the same level of abstraction [72]. A transformation that derives source code from a UML class diagram is a vertical model transformation. Model refactorings are an example of horizontal endogenous model transformations. A *unidirectional* model transformation can only transform a source model to a target model, whereas a *bidirectional* model transformation can take as input models of target type and produce models of source type [50].

To implement common model transformation tasks, a number of specialized transformation languages have been proposed such as ATL [85][175], Query/View/Transformation (QVT) [140], Tefkat [108], and Epsilon [178]. Although the problem domain of these languages is same, they differ in the employed programming paradigms (declarative, imperative, object-oriented, functional, etc.) [86].

1.2 Antipatterns

An *antipattern* can be defined as a potentially bad solution to a commonly occurring design problem. Antipatterns are the opposite of design patterns, which represent good design practices, and result in high quality software. Antipatterns may occur when inexperienced designers attempt to incorporate design patterns in an incorrect context. Presence of antipatterns in a given design alerts the modeler of possible design flaws. Refactoring an antipattern instance changes the flaws into a healthy solution. Several antipatterns have been documented in the literature such as *blob* (also known as *god class*), *functional decomposition*, *swiss army knife*, *poltergeists* and *spaghetti code* [36]. Bad design practices at the code level are known as *bad smells* in the literature. Bad smells are fine-grained, and can be detected from code; on the contrary, antipatterns are coarse grained, and can be detected at the design level [70]. Several bad smells exist in the literature such as *data class*, *shotgun surgery*, *long method*, and *lazy class* [71]. In this thesis, we focus on bad design practices at the requirements level, i.e. use case modeling antipatterns, and propose a model transformation approach for their detection and refactoring.

An antipattern provides means to change a fallacious solution to a proper one by providing some key information. In the context of use case modeling, an antipattern will describe an unsound description, and its potential harmful consequences downstream in the development process. An antipattern description will also explain why such an unsound structure may have seemed appropriate in the first place. Most importantly, an

antipattern description will describe the appropriate structure that should be used instead. The information used to describe an antipattern should be obtained from actual practice. Antipatterns are usually described using a template. The templates presented in [18][19][20][22] were specifically designed to describe use case antipatterns.

1.3 UCM Notation

A UCM consists of one or more paths each of which represent a use case scenario. A path starts at a *start point* (filled circle) and ends at an *end point* (bar). The actions performed by the system or use case actor along these paths are *responsibilities* (cross). These responsibilities can be bound to *components*—*actors, agents, teams, objects and processes*.

An actor component (rectangle including a stickman) represents a stakeholder who is associated with the system through a number of usage scenarios. Software agents in agent-oriented systems can be represented by the agent component (rectangle with a dark border). Teams (rectangle) represent high level abstract components that can be further decomposed into multiple levels of other component types. However, objects (box with rounded corners), which represent instances of a class, cannot be further decomposed. Processes (slanted rectangle) are executing components of a system and may include object components.

An *OR-fork* divides a path into one or more alternative paths based on a guard condition. Concurrent paths emerge from *AND-forks* (bar). Common paths are merged by *OR-joins* and concurrent paths are synchronized by *AND-joins* (bar). Erroneous situations that may stop the flow of a path are represented by *failure points* (ground). *Timers* (clock) express the amount of time to wait before a path can progress further. A *waiting place* (filled circle and bar) allows a path to wait for another path to finish before it can continue.

Stubs (diamond) are containers for nested maps. Stubs are useful for refactoring complex UCMs via modularization. The interested reader may refer to Buhr and Casselman's [39] book on UCMs for more details on its notation. Figure 1 summarizes the UCM notation.

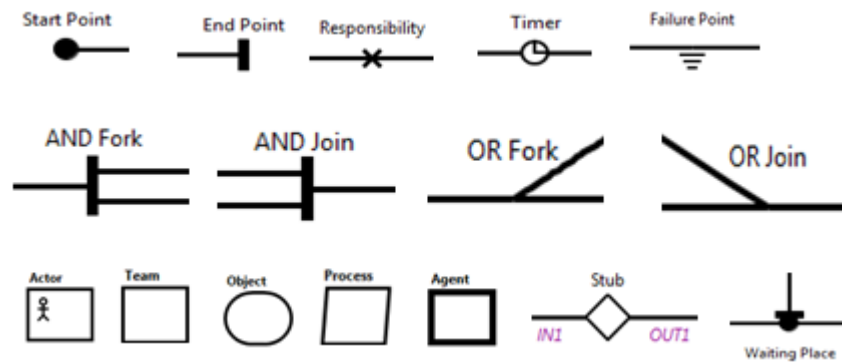


Figure 1: UCM notation

1.4 UML 2 AD Notation

An *activity* in an AD is a directed graph comprising of *activity nodes* and *activity edges*. The Object Management Group's (OMG) UML superstructure specification [141] defines three types of activity nodes—*action nodes*, *object nodes* and *control nodes*. *Control flow* is an activity edge that represents the transitions between activity nodes. Action nodes exchange messages with each other through the *object flow* edge. Both control and object flows are represented as an arrow.

Action nodes (box with rounded corners) represent the actions to be performed by the system being modeled within a particular context. The exchange of messages between actions is modeled by object nodes. Control nodes coordinate the execution of an AD. The flow of an activity starts at an *initial node* (solid circle) and stops at a *final node* (solid circle surrounded by hollow circle). Concurrent flows of control emerge from *fork nodes*. Alternate flows of control initiate from *decision nodes*. *Join nodes* synchronize concurrent flows, and *merge nodes* combine alternate flows.

Activity partitions or *swimlanes* are regions on an activity surrounded by parallel lines, either horizontal or vertical. They group related nodes together, represent organizational units such as classes [159] and may nest other partitions. A *structured activity node* (dashed box with rounded corners) is defined as “an executable activity node that may have an expansion into subordinate nodes as an ActivityGroup” [141]. *ActivityGroup* refers to an abstract meta-class in the UML 2.2 metamodel, which groups a set of activity nodes and edges [185]. Activity partitions and structured activity nodes

inherit from this metaclass. The interested reader may refer to the OMG UML 2.2 specification [141] for more details on AD notation. Figure 2 summarizes the UML 2 AD notation.

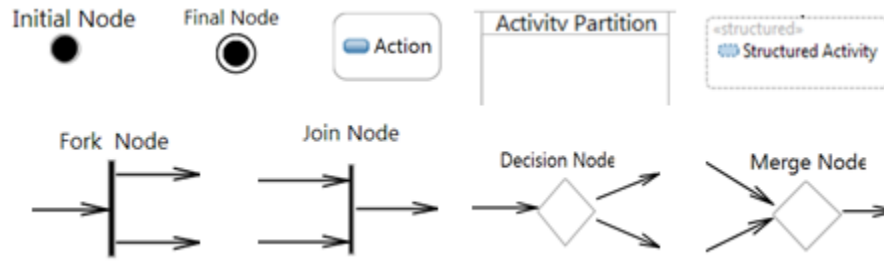


Figure 2: UML 2 AD notation

1.5 UML 2 SD Notation

SDs model OO system scenarios as a sequence of interactions between system objects, represented as *lifelines*. Actors (users or external systems) that interact with the system objects are also depicted as lifelines. SD can model scenarios at different levels of detail; in a high-level SD, a lifeline can be a system, subsystem, or component, whereas detailed SDs include *boundary*, *controller* and *entity* objects as lifelines.

The different lifelines communicate by passing *messages* to one another. Messages are represented as arrows connecting a source and target lifeline. The different types of messages in the UML 2 notation are *synchronous*, *asynchronous*, *create*, *destruct* and *reply*. If a source lifeline sends a synchronous message, it waits for a response from the target lifelines. In asynchronous messaging, the source continues its execution after

sending a message. Create messages depict initialization of target lifelines, whereas destruct messages depict their destruction.

InteractionUse allows a SD to reference another one; therefore, enables multiple SDs to share common interaction sequences. *Gates* are connection points which pass messages to or from a SD. External messages coming through a gate initiate execution of a SD. Terminating messages in a SD are passed out through a gate. *State invariants* indicate the state of a lifeline at a particular point of time in a SD's execution. A state can indicate the value of an attribute or variable, or constraints on the lifeline.

Fragments are regions on a SD that group related messages together. One or more *operands* form the body of a fragment. Each operand has a *guard*, a boolean expression, which must evaluate to true for the operand to execute. Twelve different types of fragments are defined in the UML 2 specification. Alternate flows in a scenario are represented by the *alt* fragment. The *par* fragment represents concurrent execution of operands. The termination of a *break* fragment indicates that remainder of the messages in its enclosing fragment, or SD, will not execute. The *loop* fragment can depict repeated behavior in a scenario. The interested reader may refer to the OMG UML 2.2 specification [141] for remainder of the fragments, and more details on the SD notation. Figure 3 summarizes the UML 2 SD notation.

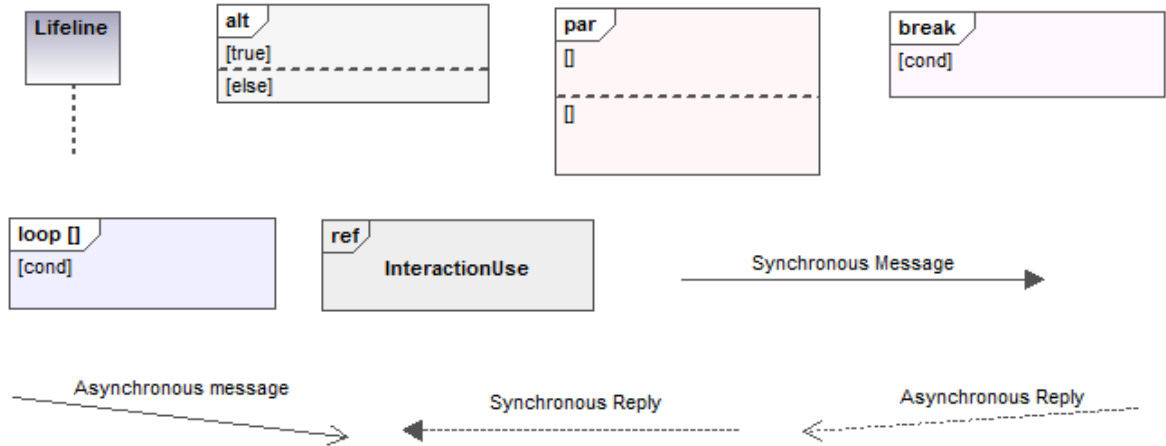


Figure 3: UML2 SD notation

1.6 Atlas Transformation Language

ATL is a model transformation language that provides declarative and imperative constructs for implementing model-to-model transformations. The input to an ATL transformation includes one or more source models. The output of an ATL transformation is, typically, one target model. Figure 4 illustrates an ATL transformation pattern. In the pattern, a source model Ma is transformed into a target model Mb according to a transformation definition $mma2mmb.atl$, written in ATL. The transformation definition is also regarded as a model. The source and target models, and the transformation definition conform to their metamodels MMa , MMb , and ATL respectively. The metamodels conform to the MOF metamodel [142].

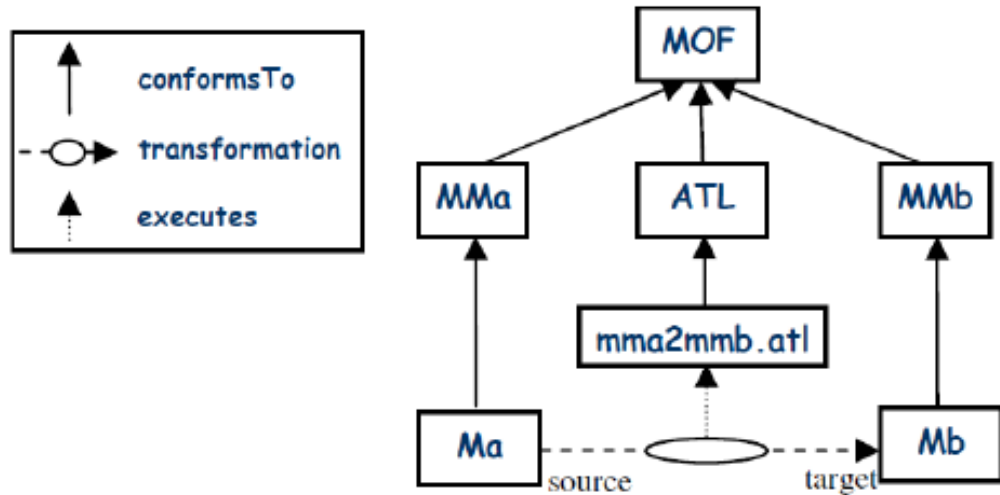


Figure 4: Overview of ATL model transformations

1.6.1 ATL Rules

An ATL model transformation is specified in a *module*, which contains a set of *rules*. ATL allows developers to specify two types of rules, *declarative* and *imperative*. Declarative rules are also referred as *matched* rules in ATL. Imperative rules must be explicitly invoked in an ATL module by the programmer, whereas matched rules are implicitly called at runtime. Rule *AtoB* in Table 1 is an example of a *matched* rule. In *AtoB* rule, *s* refers to an object of type *A*, and *t* refers to an object of type *B*. *A* and *B* are metaclasses defined in the source and target metamodels, respectively, of *AtoB*'s enclosing module. In the mapping statement “*b1* ← *s.a1*”, *a1* and *b1* refer to attributes of the classes *A* and *B*, respectively.

ATL includes two kinds of imperative rules, *lazy* rules and *called* rules. They differ in their implementations, but their functionalities are identical. Both are defined within

the context of their corresponding module; thus, they are invoked using the *thisModule* keyword, which is equivalent to Java's *this* keyword. The definition of a *lazy* rule does not include formal parameters; however, when they are invoked the source object must be passed as an actual parameter. The mapping statement "*b2* ← *thisModule.CtoD(s.a2)*" in *AtoB* (see Table 1) invokes the lazy rule *CtoD*. The actual parameter passed to *CtoD* is *s.a2*.

Called rules may or may not contain formal parameters. They can be further classified into *mapping* and *non-mapping*. The former contains a mapping from a source instance to target instance, whereas the latter contains none. For instance, the called rule *EtoF* in Table 1, contains a *to* clause "*t: F*", whereas the called rule *PrintF* does not contain a *to* clause. The *do* block, in called rules, allows developers to specify imperative statements. For example, *f.println()* is an imperative statement in rule *PrintF* (see Table 1). Imperative statements are optional in *matched* and *lazy* rules, whereas they are mandatory in *called* rules. In mapping called rules, the last statement of the *do* block must return the target object. For instance, the statement "*t;*" returns the target object in rule *EtoF*.

Table 1: Example of matched, lazy, mapping called, non-mapping called rules

Matched Rule	Lazy Rule	Mapping Called Rule	Non-Mapping Called Rule
<pre>rule AtoB { from s : A to t: B (b1 <- s.a1, b2 <- thisModule.CtoD(s.a2), b3 <- thisModule.EtoF(s.a3)) }</pre>	<pre>lazy rule CtoD { from s : C to t: D (.....) }</pre>	<pre>rule EtoF(s: E) { to t: F (.....) do { PrintF(t); t; } }</pre>	<pre>rule PrintF(s: F) { do { f.println() } }</pre>

1.6.2 ATL Helpers

Helpers are the ATL equivalent of methods in the OO paradigm. Helpers are written in the context of a source metaclass. They enable querying of source model objects. Helpers differ from rules since they do not create target model objects. Parameter-less helpers are referred as *attributes*. Table 2 shows an example of a helper, and an attribute. The helper *findB*, defined in the context of metaclass *A*, is defined to find object *b*, of type *B*, in attribute *a1*. The attribute *isPositive*, defined in the context of *A*, is defined to determine whether the value of attribute *a2* is greater than zero or not.

Table 2: Example of ATL helper and attribute

Helper	Attribute
<pre>helper context A def: findB(b: B) : B = self.a1->any(i i = b);</pre>	<pre>helper context A def: isPositive : Boolean = self.a2 > 0;</pre>

1.6.3 ATL Execution Modes

ATL modules can execute in two modes, *default* and *refining*. Default mode is the normal execution mode of ATL transformations and it is specified by the *from* keyword. Default mode is intended for exogenous model transformations; therefore, the UCM to

AD, and UCM to SD transformation rules in Section 4.2 and Section 5.2, respectively, are implemented in default mode. Refining mode is applicable only for endogenous model transformations; therefore, some of the use case model refactorings presented in Section 3.1 are implemented in refining mode. Typically in model refactoring, only few objects of the source model undergo changes, whereas the remaining objects are copied into the target model. Refining mode allows developers to define rules only for those objects that need to be transformed; the remaining objects will be implicitly copied into the output model. Therefore, refining mode is an excellent choice for implementing model refactorings. However, the use of refining mode is limited as it does not allow developers to specify imperative rules. Consequently, remainder of the use case model transformations presented in Section 3.1 are implemented in default mode.

Table 3 shows an example of modules in default mode and refining mode. Module *A* is defined in default mode, whereas module *B* is defined in refining mode.

Table 3: Example of modules in default mode and refining mode

Default mode	Refining mode
<pre>module A; create OUT : UML from IN : UML;</pre>	<pre>module B; create OUT : UML refining IN : UML;</pre>

1.6.4 ATL Module Superimposition

Module superimposition [183] is mechanism that enables the reuse of generic rules across multiple ATL modules. Let module *A* contain the set of rules $R_A = \{a_1, a_2, a_3\}$, and module *B* contain the set of rules $R_B = \{b_1, b_2, b_3\}$. If *B* is superimposed on *A*, then

the superimposed module S will contain set of rules $R_S = R_A \cup R_B = \{a_1, a_2, a_3, b_1, b_2, b_3\}$. If a_1 and b_1 have the same name, then b_1 will overwrite a_1 resulting in $R_S = \{a_2, a_3, b_1, b_2, b_3\}$. The developer must make sure that superimposition results in a *confluent* [129] set of rules, in which no two rules must be applicable on the same source object.

Although refining mode is ideal for defining model refactorings, it cannot be used in situations where the developer wants to write imperative code. This forces the developer to implement his desired model refactorings in default mode. In model refactoring, a large number of model objects remain unchanged, and must be copied from the source model into the target model. In refining mode, this copying is performed automatically by the ATL virtual machine. On the other hand, default mode requires the developer to define trivial rules for copying each unchanged model object. This becomes tedious when implementing a large suite of model transformations, and results in code (rule) duplication across the different modules. This problem can be averted using module superimposition, which allows the implementation of reusable modules. Therefore, module superimposition is an alternate way to efficiently implement model refactorings. Module superimposition was used in every default mode model transformation presented in Section 3.1. ATL rules for copying use case model objects were defined in separate modules, and superimposed on relevant model transformations.

1.7 Research Question

The main research question that we aim to answer in this thesis is the following:

How can model transformation techniques to be used to improve the Functional Requirements Specification (FRS) activity of a software process?

The research question will be answered in two folds by defining and implementing model transformations for:

- Improving the quality of use cases
- Deriving high-level design models (ADs and SDs) from UCM scenario specifications

The proposed model transformations will present a case for software developers to embrace the notion of model transformation in the context of FRS.

1.8 Thesis Objectives

The objectives of this thesis are as follows:

1. Propose a fast, efficient, and scalable technique for improving the FRS activity. By FRS, we refer to use case modeling and scenario modeling.
2. Implement the proposed technique using various tools
3. Demonstrate the feasibility of the proposed technique on case studies that pertain to real-world software systems
4. Compare the results of the proposed technique with previous work in order to identify its significance.
5. Propose and implement a framework for enabling thorough verification of the proposed technique.

1.9 Research Methodology

The research methodology followed in thesis is as follows:

Literature Review

A literature review was performed to study the existing techniques for improving FRS. Throughout this work, the literature was rigorously reviewed for understanding the related work done in this domain.

Propose Model Transformation Techniques

After the initial analysis of existing techniques, a new model transformation technique for improving FRS was proposed. We improve FRS by enhancing the quality

of use case models in an efficient and scalable manner, and by automatic derivation of high-level design models from UCM scenario specifications.

Implementation of the Proposed Techniques

The tools used for the implementation of the proposed model transformation techniques are:

1. Integrated Development Environment – Eclipse Indigo [176]
2. Integrated Development Environment – Visual Studio 2012
3. Model Transformation Language – ATL [85][175]
4. Eclipse based UCM modeling tool – jUCMNav [87]
5. Eclipse based UML modeling tool – UML 2 Tools [177]
6. UML modeling tool – Altova UModel 2008 [181]

Evaluation of the proposed techniques

After successful implementation of the proposed model transformation technique, its effectiveness was assessed on case studies.

The proposed antipatterns based use case quality improvement technique was evaluated on the use case models of MAPSTEDI (Mountains and Plains Spatio-Temporal Database Informatics) system [116]. The MAPSTEDI system is a distributed database system that integrates biodiversity data collections from three sources, the University of Colorado Museum, the Denver Museum of Nature and Science, and the Denver Botanic Gardens. The integrated database contains 285,000 biological specimens. The system will

allow geocoders to analyze biodiversity data in the southern and central Rocky Mountains. A map based GUI is provided by MAPSTEDI to allow users to geographically reference the specimens.

The UCM to UML 2 AD, and the UCM to UML 2 SD, model transformations were validated using the UCM of an Elevator Control System (ECS), which is available at [8]. The UCM was adapted from “*Designing Concurrent, Distributed and Real-Time Applications with UML*” [74]. Another case study which covers the entire UCM notational set is also presented for illustrating the UCM to UML 2 AD model transformation. The framework for thorough validation of the proposed techniques is validated on the UCM to UML 2 AD model transformation.

Conclusion

The conclusion of the thesis summarizes the research performed, and its benefits. In addition, future research directions in the area of MDE are discussed.

Thesis Writing

This research was documented in a thesis form which was rigorously updated based on inputs from the thesis advisor, Dr. Mohamed El-Attar, and committee members, Dr. Mahmoud Elish and Dr. Sajjad Mahmood. Finally, it was submitted to the Deanship of Graduate Studies (DGS) once approved.

1.10 Thesis Outline

The remainder of this thesis is organized as follows. Chapter 2 presents related work on quality improvement of use case models, antipatterns, model transformation based model refactoring, verification of model transformations, and transformation of UCMs to other modeling notations. Chapter 3 presents a model transformation approach for improving the quality of use case models, and presents a real-world case study to evaluate the effectiveness of the proposed approach. The case study pertains to the use case model of a bio-diversity database system. Chapter 4 defines a model transformation for deriving UML 2 ADs from UCMs, and presents two case studies to illustrate the transformation. The first pertains to an Elevator Control System (ECS), and the second pertains to a mock system. Chapter 5 defines mappings from UCM constructs to UML 2 SD notation, and defines a model transformation to implement the mappings. A more refined version of the ECS case study is used to demonstrate the transformation. Chapter 6 proposes a suite of mutation operators for fault based verification of ATL transformations, and evaluates their effectiveness on the UCM to UML 2 AD model transformation. Chapter 7 concludes the thesis and discusses future work.

CHAPTER 2

LITERATURE REVIEW

This chapter presents a survey of the related literature on use case quality improvement techniques, antipatterns, model transformation based model refactoring, verification of model transformations, and transformation of UCMs to other modeling notations.

2.1 Use Case Quality Improvement

2.1.1 Using Inspection, Guidelines, and Templates

Use case inspection is a strongly suggested method for ensuring consistency of use case models [17][103][163]. A checklist-based inspection technique was presented in [11] based on best practices provided in [17][163]. The usage of this technique is limited as it requires a great deal of use case modeling expertise. Linguistic techniques have been suggested for detecting defects in use case descriptions [63]. However, they are not adequate for ensuring correctness and consistency of requirements.

Styling and content related guidelines which enhance consistency in use case descriptions are presented in [3]. Experimental evaluation revealed that the guidelines in [3] do not necessarily improve the correctness of use case descriptions [49]. A

“conversational” style for authoring use case descriptions is advocated in [187]. Special styles for capturing business rules [13], and user interface requirements [30][46], in use case descriptions have also been devised. A standard for documenting use cases of embedded systems is proposed in [149].

A list of critical use case modeling mistakes which led to project failure or delay is given in [43]. Many common use case modeling mistakes made by inexperienced practitioners are presented in [111]. The mistakes can be detected early in the development cycle by performing checklist based reviews. This technique is practical for small scale systems, which have few number of use cases. However, for complex systems, containing thousands of use cases, checklist based reviews are cumbersome and prone to human error. Therefore, an automated technique for improving the quality of use case models is presented in this thesis. The heuristics behind the use case modeling mistakes listed in [17][111][163], and other best practices from the literature, such as those presented in [4][62][66][75][98][99][126][146][157], are incorporated in the antipatterns approach.

Templates for writing high quality use case descriptions are described in [32][45][78][83][121]. Anda et al. [12] have shown that using templates significantly improves understandability of use cases. A machine readable structure for use case authoring that ensures consistency between use case diagrams and use case descriptions is presented in [21].

2.1.2 Using Automated Verification Tools

Ryndia et al. [160] developed a use case verification tool *SusanX*, which relies on detailed knowledge of the system's domain. The tool may not be useful in the early stages of a software development cycle because detailed domain knowledge is unavailable. Moreover, the tool does not consider relationships between use cases. The approach presented in this thesis does not require domain knowledge; therefore, it can be applied early in the development cycle. Furthermore, it is based on the antipatterns approach which considers use case relationships that are not handled by *SusanX*. Berenbach [29] described heuristics for constructing verifiable, understandable, correct, complete and consistent use case models, and incorporated them in an analysis tool, *Design Advisor*. Some of the heuristics can be automatically verified, while others require manual effort. However, the tool does not perform any refactorings itself. The heuristics presented in [29] are a subset of the heuristics incorporated in the antipatterns approach.

The Requirements Use Case Tool [123] is a web based application that searches a given use description for a set of risk indicators, which may negatively impact use case quality attributes. The approach presented in this this does not need to be used exclusively. In fact, it is recommended that this approach be used in addition to other quality improving techniques, such as use case templates and authoring guidelines, in order to leverage their collective advantages.

2.1.3 Use Case Refactoring

Refactoring is the process of enhancing the structure of a software artifact without changing its intended behavior [71]. Refactoring was first introduced by Opdyke [145]

for OO source code. The concept of source code refactoring has been extended for UML models, including use case models, in order to improve their quality.

The notion of use case refactoring was first considered by Butler et al. [40] within the context of product line evolution. Use case refactorings were shown to document product line variability and evolution in [188]. Yu et al. [191] listed 10 use case refactoring rules based on “episodes”. Source code refactorings from [145] were extended to use case models in [158]. Five types of primitive refactorings on use case modeling constructs were defined—*create*, *delete change*, *move* and *decompose*. These refactorings were later implemented using a metamodel specification and incorporated in a graphical tool [189]. In [95], five other primitive use case refactorings, *decomposition*, *equivalence*, *generalization*, *merge*, and *delete*, were defined in the context of service oriented architectures. In this thesis, several composite use case refactorings, which are sequences of primitive use case refactorings, are defined and implemented. Moreover, the refactoring presented in [40][95][191] are already incorporated in the antipatterns approach.

Refactorings that apply on use case descriptions are described in [80] and [155]. The antipatterns based approach incorporates the concept of refactoring use case models; it can be used in tandem with the use case description refactorings described in [80] and [155].

2.2 Antipatterns

2.2.1 Impact of Antipatterns on Quality Attributes

The effect of antipatterns and bad smells on quality aspects of software has been empirically investigated in the literature. In [1], it was revealed that a combination of the *god class* and *spaghetti code* antipatterns adversely affected understandability. Deligiannis et al. [51][52] showed that the presence of *god classes* in a design makes maintenance activities difficult, and deteriorates the design. Refactoring of *god classes* revealed better design comprehensibility in [56]. In [110], it was shown that classes having antipatterns *god class*, *god method*, and *shotgun surgery* were more fault prone than other classes. Olbrich et al. [143] analyzed the historical data of two large scale open source systems, and concluded that classes having antipatterns *god class* and *shotgun surgery* changed more frequently than other classes. Romano et al [156] showed that classes containing antipatterns are more change-prone than classes free of antipatterns. In recent work, Khomh et al. [100] analyzed several releases of four software development tools, and concluded that classes with antipatterns are more change-prone and fault-prone than others. Hence, there is need for antipattern detection and subsequent refactoring.

2.2.2 Antipattern Detection

Early antipattern detection and correction significantly improves software quality. This has prompted several researchers to propose techniques for detecting *design* level antipatterns. Wieman [186] developed a heuristics based detection tool for design level antipatterns, violations of design principles and code smells. Empirical evaluation of the

tool on open source projects revealed that design antipatterns exist in OO software. A metrics based approach for detecting design antipatterns is presented in [70]; this approach considers structural and behavioral aspects of design, whereas the tool in [186] considers structural aspects only. A tool that automatically generates algorithms for detecting design antipatterns is presented in [132].

Ballis et al. proposed a language for specifying both design patterns and antipatterns in [24], and defined rules for their detection in [23]. A numerical analysis based technique accurately distinguished between antipatterns and non antipatterns at the design level in [144]. A logic based detection approach, which used Prolog predicates, was proposed in [174], and successfully validated on open source projects. Machine learning techniques, Bayesian networks [101][170], and support vector machines [115], have also been used in the literature to detect design level antipatterns. Other detection approaches found in the literature are based on inspection [179], heuristic search [91], predicate logic [5], metamodeling [182], visualization [54][107][171], and metrics [117][138].

In [61], model checking rules were derived from several antipattern descriptions to detect undesirable properties in class diagrams. Cortellessa et al. [47] showed how Object Constraint Language (OCL) queries can be used to detect the *blob* antipattern in UML component, sequence, and deployment diagrams. Automatic detection of performance antipatterns that pertain to architectural models is performed in [16][48][180]. This thesis focusses on antipatterns within the requirements engineering context, more specifically at the functional requirements specification (use case modeling) level. Furthermore, the

refactorings proposed in this thesis are performed at a much earlier phase of a software process (requirements engineering) compared to existing literature (architecture, design and code levels).

Liu et al. [113] presented an approach to detect overlapping use cases based on SDs and state-charts. The proposed approach cannot be applicable early in a software process as SDs and state-charts may not be specified. In earlier work [22], use case antipattern detection was automated by describing antipattern designs as constraints using OCL. However, model transformation provides a more powerful mechanism to detect antipatterns, which was otherwise not possible to detect when written using OCL. Therefore, this thesis proposes a model transformation approach for antipattern detection and refactoring.

2.3 Model Transformation based Model Refactoring

Model transformations can greatly ease several software development activities including model refactoring [169]. This has prompted several researchers to implement model transformations for performing model refactoring.

Mens stated that one of the ways to perform model refactoring is the application of design patterns [127]. Model transformations, implemented in XSLT, were presented in [190] for automatically applying design patterns on UML class diagrams. XSLT was used to implement reusable model transformation in [102]. Similar to XSLT, ATL also allows reuse via the concept of module superimposition.

Demuth et al. [53] showed that XSLT based model transformations can be used for deriving SQL schemas from UML class diagrams, and reverse engineering UML class diagrams from code. However, the usage of XSLT for implementing model transformations is limited since it is fundamentally a declarative language. On the other hand, ATL is a hybrid language which permits imperative programming styles, in addition to declarative. Therefore, the model transformations presented in this thesis are implemented in ATL due to its inherent versatility and reusability (via module superimposition).

Other refactoring techniques using model transformation were developed but not for use case modeling. Zhang et al. developed a model transformation engine to perform generic and domain specific model refactorings [193]. Other model transformations have been defined for refactoring UML class diagrams [69][97][109][118][119][133][152][153], ADs [57][58], state-charts [69], KAOS models [42], feature models [164], Alloy object models [120], software architectures [81], executable UML models [55], and Java source code [137].

2.4 Verification of Model Transformations

Model transformation testing is gaining interest within the MDE community, as the size and complexity of model transformation programs grow. Testing model transformations exhibits many challenges [25][26]. Two important challenges that have been investigated in the literature are the efficient generation/selection of test cases, and the definition of an oracle function to assess the validity of the transformed models.

Fleurey et al. [68] investigated the problem of test data generation for model transformations, and proposed the use of partition testing to define test criteria to cover the input metamodels. Lamari [106] used a functional testing approach based on a data partitioning technique that focuses on the structure of models in order to take into account the structural aspect of models when generating input test models. Fiorentini et al. [65] have proposed a uniform framework for treating metamodels, model transformation specifications, and the automation of test case generation. The proposed technique in [65] is based on a black-box testing approach of model transformations to validate their adherence to given specifications. White-box test model generation approaches for ATL model transformations have been proposed in [76] and [125]. Another white-box approach, which is based on static analysis of structural information in model transformations, is presented in [134]. A gray-box testing technique has also been used by Bauer and Küster [27] for model transformations. Sen et al. [167] presented a tool for semi-automated generation of test models from knowledge such as requirements, known faults, and existing inputs.

Lin et al. [112] have presented a framework for test case creation and execution with a particular focus on the problem of model comparison of expected and actual outputs models. Wang et al. [184] have proposed structural testing of model transformations using the metamodel coverage criteria. The types of faults that can occur while implementing a transformation rule are described in [105]; test case generation for model transformations should focus on detecting such faults. The unit testing technique has been demonstrated on QVT-O model transformations in [44]. Cabot et al. [41] have

used OCL invariants, derived from transformation rules, to verify model transformations written in QVT-R and Triple Graph Grammars. Küster et al. [104] proposed a framework for automated testing of model transformation chains.

The authors in [73] and [124] have used the test driven approach to implement model transformations. Giner and Pelechano [73] have defined a test-driven method to capture requirements for transformations in such a way that guides the development and documentation of model transformations. Requirements were expressed by means of test cases that can be automatically validated. McGill and Cheng [124] have extended the JUnit testing framework with assertions that simplify the testing of model transformations. These extensions are implemented in a tool called *Jemtte*.

Techniques for assessing the quality of model transformation test cases have been described in [28] and [67]. Test oracles, which are strategies for determining whether a test case passes or fails, for model transformations have been discussed in [90] and [135]. In recent work [173], the application of model comparison techniques has been investigated for defining model transformation test oracles. Automated tools for the generation of test input have been presented in [35] and [77]. For a detailed survey on the diverse approaches to model transformation verification, the reader is invited to refer [166].

Mottu et al. [136] introduced the application of mutation testing to model transformations. The authors [136] have identified four semantic classes of faults (navigation, filtering, output model creation, and input model modification) for model transformations, and they have defined a set of generic mutation operators to cover these

fault classes. The effectiveness of these mutation operators was demonstrated in [168] by automatic test model generation followed by mutant execution. These generic mutation operators can be adapted for different model transformation languages. However, there is a need to define mutation operators that can capture model transformation programming language specific characteristics. Therefore, in this thesis, a suite of mutation operators are proposed for ATL.

2.5 UCM transformations

To allow traceability of functional requirements, several studies have proposed mappings from UCMs to other modeling notations. Bordelau and Cameron [33] defined a systematic and traceable way of deriving Message Sequence Chart (MSC) scenario models from UCMs. Miga et al. [130] extended Bordelau and Cameron's [33] systematic procedure and implemented a UCM to MSC transformation using a prototype tool. He et al. [79] illustrated the generation of MSCs from UCMs using an automated tool. Amyot et al. [7] implemented a model transformation in XSLT for deriving MSCs from UCMs. Bordeleau and Buhr [34] proposed modeling steps from UCM to ROOM state machines [165]. These steps help in bridging the conceptual gap between the notations, and enables traceability from detailed design to scenarios. A method for deriving SDL diagrams [60] from UCMs was proposed by Sales and Probert [161]. In [148], an algorithm was proposed for generating software performance models from UCM specifications. A prototype tool was presented in [88] for automatic derivation of UCMs from natural language use case descriptions. Martínez [122] showed how state-charts can be

synthesized from the combined information provided by UCMs and UML collaboration diagrams. Zeng [192] defined a transformation from UCMs to *core scenario models*, which allow the quick generation of software performance models. Amyot and Mussbacher [10] proposed an extension of UML 1.3 with UCM core concepts for the purpose of introducing a new “UCM View” to the existing set of UML views. To date, the proposed “UCM View” is not a UML standard. Hence, there is a need for a mapping between these distinct notations.

To the best of our knowledge, no attempt has been made to propose mappings between UCM and UML 2 notations. The mappings and model transformations presented in this thesis can facilitate the transition from requirements to high-level design. This thesis has also not only proposed mappings between the UCM and UML modeling notations but also suggests automation of mappings. Automation will enable requirements traceability, as well as reduce the effort required to derive detailed design from scenarios represented as UCMs.

CHAPTER 3

A MODEL TRANSFORMATION APPROACH

TOWARDS REFACTORING USE CASE MODELS

BASED ON ANTIPATTERNS

In this chapter, we present use case antipatterns and define model transformations for their refactorings. The use case antipatterns are adopted from the work of El-Attar and Miller [18][19][20][22]. The taxonomy of antipatterns presented in [18] was developed via a systematic review of current practices for constructing high quality use case models. A large subset of these antipatterns prescribes refactorings that can be carried out using model transformation. Table 4 provides a summary of use case antipatterns and the corresponding refactorings that will be implemented using model transformation.

Table 4: Use case antipatterns and their respective refactorings

Use Case Antipattern	Refactoring
a1. Accessing a <i>generalized concrete</i> use case	r1. Concrete to Abstract r2. Drop Actor-Generalized UC Association
a2. Accessing an <i>extension</i> use case	r3. Drop Actor-Extension UC Association r4. Directed Actor-Extension UC Association
a3. Using <i>extension/inclusion</i> use cases to implement an <i>abstract</i> use case	r5. Abstract Extended UC to Concrete r6. Inclusion to Generalization
a4. Functional Decomposition: Using the <i>include</i> relationship	r7. Drop Functional Decomposition r8. Drop Functional Decomposition having Inclusion
a5. Functional Decomposition: Using the <i>extend</i> relationship	r9. Split Extension UC r10. Extension to Generalization
a6. Multiple <i>generalizations</i> of a use case	r11. Generalization to Include
a7. Use cases containing common and exceptional functionality	r12. Drop Inclusion r13. Drop Extension
a8. Multiple actors associated with one use case	r14. Generalize Actors r15. Split UCs
a9. An association between two actors	r16. Drop Actor-Actor Association
a10. An association between use cases	r17. Drop UC-UC Association
a11. An unassociated use case	r18. Drop Unassociated UC
a12. Two actors with same name	r19. Rename Actor
a13. An actor associated with an unimplemented <i>abstract</i> use case	r20. Abstract to Concrete r21. Add Concrete UC

A source use case model that contains one or more instances of a use case antipattern is provided as input to its suitable model transformation. A model transformation detects the model elements involved in an antipattern, and performs appropriate refactoring; thus, resulting in a target use case model. This target use case model is free of the use case antipattern present in the source use case model. The transformations are endogenous, horizontal, out-place, one-to-one, unidirectional model transformations. The metamodel used for implementing the model transformations is the

Eclipse Modeling Framework's (EMF) [37] implementation of the OMG UML 2.0 specification [141]. The source and target use case models conform to this EMF metamodel; therefore, the transformations are endogenous. Since refactoring does not alter the level of abstraction in which the source is expressed, the model transformations fall in the horizontal category. The model transformations are implemented using ATL [86][175]. In ATL, the source and target models of a given model transformation are distinct entities; therefore, the transformations are out-place. Because the input to the transformations is one source, and the output is one target, the model transformations are one-to-one. An ATL module cannot be used to reverse engineer the source from the target; therefore, the model transformations are unidirectional. The ATL source code is available to the interested reader for download at [92].

It is important to note that the existence of an antipattern in a use case model, by definition, does not prove the presence of a defect. The detection of an antipattern in a use case model will only prompt the modeler to reconsider their design due to the likelihood of costly work downstream resulting from the current design. Upon evaluating the use case model instance based on the information provided by the corresponding antipattern description, the modeler will then determine whether their design is indeed defective or not. If the design is considered defective, then refactoring measures are undertaken to improve the quality of the model; otherwise, no further action is required.

The remainder of this chapter is organized as follows. Section 3.1 describes the proposed model transformation approach for executing use case refactorings. Section 3.2

demonstrates the feasibility of the approach using a case study that pertains to a biodiversity database system. Finally, Section 3.3 evaluates the results of the case study.

3.1 Use Case Modeling Antipattern Refactorings

a1. Accessing a *generalized concrete* use case

This antipattern occurs when an actor is associated with a *generalized* use case in order to enable indirect access to a framework of services, which are implemented by *specialized* use cases. A *generalized* use case is often incomplete because it contains parts of common behavior required by the *specialized* use cases. Therefore, initiation of such a *generalized* use case will result in incomplete meaningless behavior.

r1. Concrete to Abstract

This refactoring converts the *generalized* use case to *abstract*. The semantics of *abstract* use cases are same as the semantics of an *abstract* entity in the OO paradigm. Setting a use case as *abstract* indicates that it cannot be solely performed. Therefore, one of the *specialized* use cases will be performed. This guarantees that a complete and meaningful service will be delivered to the actor. A given use case is involved in this antipattern if it:

- is a *concrete generalized* use case
- neither *includes* nor *extends* any use case
- neither *included* nor *extended* by any other use case
- is directly or indirectly associated with an actor

The rule *ConcreteToAbstract* in Listing 1 checks the above detection conditions for each use case in a given use case model. If a use case satisfies all the detection conditions, its *isAbstract* property is set.

```

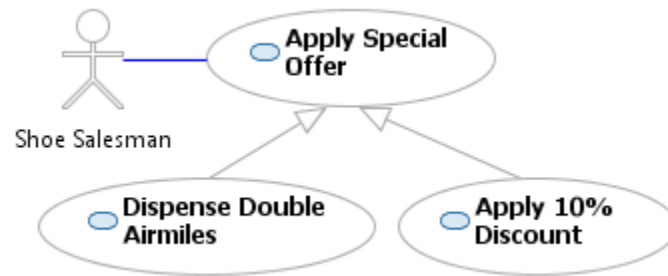
rule ConcreteToAbstract {
  from s: UML!UseCase (
    s.isGeneralization() and s.isConcrete() and not(s.isIncluder() or s.isIncluded() or
    s.isExtension() or s.isExtended()) and (s.isAssociatedWithActor() or
    s.isIndirectlyAssociatedWithActor())
  )
  to t: UML!UseCase (
    isAbstract <- true
  )
}

```

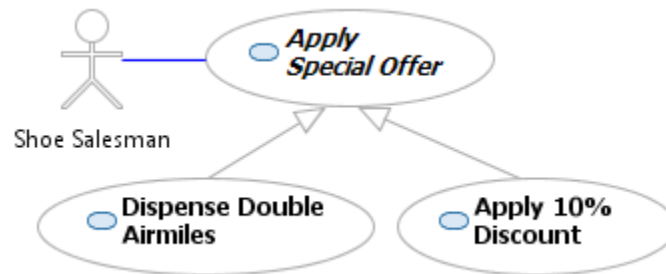
Listing 1: ATL rule for applying Concrete to Abstract refactoring

Figure 5 illustrates an example of the *Concrete to Abstract* refactoring applied on a use case model of a shoe store system. In the original use case model (Figure 5(a)), use case *Apply Special Offer* is a *generalized* use case which is *specialized* by uses cases *Dispense Double Airmiles* and *Apply 10% Discount*. The specialized use cases relate to promotional offers at the shoe store. Actor *Shoe Salesman* can apply any one of the two promotional offers on a shoe purchase by performing their corresponding use cases. Since *Apply Special Offer* is *concrete*, it can be performed exclusively by the *Shoe Salesman*. However, *Apply Special Offer* contains incomplete behavior; therefore, its exclusive execution will result in no special offer applied on a shoe sale. The application

of the *Concrete to Abstract* refactoring will set *Apply Special Offer* to *abstract*; thus, ensuring that it cannot be performed exclusively. The refactored use case model of the shoe store system is shown in Figure 5(b).



(a) Original UC model



(b) Refactored UC model

Figure 5: Example of Concrete to Abstract refactoring

r2. Drop Actor-Generalized UC Association

This refactoring replaces the association between the actor and *generalized* use case with direct associations between the actor and *specialized* use cases. It will ensure a service request is performed through one of the *specialized* use cases. Therefore, the incomplete behavior in the *generalized* use case cannot be initiated. An association is involved in this antipattern if its:

- source is an actor
- destination is a *concrete generalized* use case
- destination neither *includes* nor *extends* any use case
- destination is neither *included* nor *extended* by any use case

The rule *DropAssociation* in Listing 2 checks the above detection conditions for each association in a given use case model. If an association satisfies the detection conditions, it is deleted from the use case model. The call to rule *AddAssociations* introduces associations between the actor and *specialized* use cases.

```

rule DropAssociation {
  from s : UML!Association (
    s.isSourceActor() and s.destination().isGeneralization() and
    s.destination().isConcrete() and not(s.destination().isIncluder() or
    s.destination().isIncluded() or s.destination().isExtension() or
    s.destination().isExtended())
  )
  to drop
  do {
    thisModule.AddAssociations(s);
  }
}

```

Listing 2: ATL rule for applying Drop Actor-Generalized UC Association refactoring

Figure 6 illustrates the *Drop Actor-Generalized-UC Association* refactoring on a use case model of a shoe store system (Figure 5(a)). The association between *Shoe Salesman* and *Apply Special Offer* is incorrect, since *Apply Special Offer* contains incomplete behavior. The *Drop Generalized-UC Association* replaces this incorrect association with direct associations between *Shoe Salesman* and use cases *Dispense Double Airmiles*, and *Apply 10% Discount*.

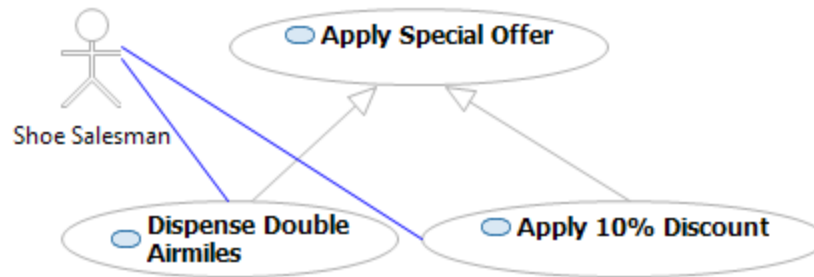


Figure 6: Example of Drop Generalized UC Association refactoring

Any one of the two refactorings can be applied to refactor antipattern a1. The refactoring r1 preserves the semantics of the original model whereas, refactoring r2 may cluster the use case model in case of several *specialized* use cases.

a2. Accessing an *extension* use case

This antipattern occurs when an actor is associated with an *extension* use case. Such a relationship is modeled in order for the actor to convey information to the *extension* use case. This is inappropriate because, an *extension* use case must be provided information from the *base* use case. A *base* use case gets the required information, an *extension* use case needs, from the actor, when it is invoked.

r3. Drop Actor-Extension UC Association

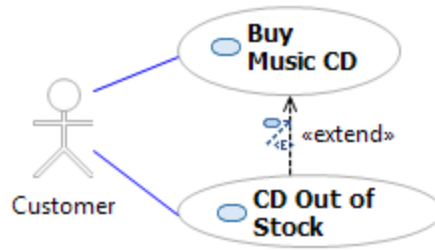
This refactoring deletes the association between an actor and *extension* use case. This ensures that the *extension* use case cannot be initiated independently, and the *base* use case provides necessary information to the *extension* use case. A given association relationship in a use case model is involved in this antipattern if its source is an actor and destination is an *extension* use case.

The rule *DropAssociation* in Listing 3 checks for such associations and deletes them when found.

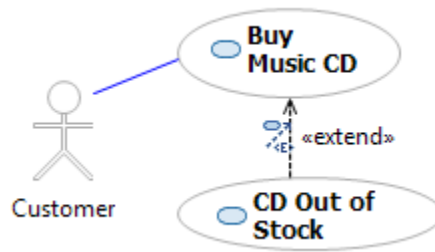
```
rule DropAssociation {
  from s : UML!Association (
    s.isSourceActor() and s.isDestinationUseCase() and s.destination().isExtension()
  )
  to drop
}
```

Listing 3: ATL rule for applying Drop Actor-Extension UC Association refactoring

Figure 7 illustrates an example of the *Drop Actor-Extension UC Association* refactoring on a use case model of a music store system. In the original use case model (Figure 7(a)), *CD Out Of Stock* is an *extension* use case that executes when a customer attempts to buy a music CD that is unavailable. Actor *Customer* is associated with this *extension* use case in order to provide it with necessary information. This association is incorrect because the *extension* use case must get the necessary information from its *base* use case *Buy Music CD*. The application of the *Drop Actor-Extension UC Association* refactoring will delete the association between the *extension* use case and *Customer*; thus, ensuring that *CD Out Of Stock* cannot be performed exclusively, and it receives necessary information from *Buy Music CD*. The refactored use case model of the music store system is shown in Figure 7(b).



(a) Original UC model



(b) Refactored UC model

Figure 7: Example of Drop Actor-Extension UC Association refactoring

r4. Directed Actor-Extension UC Association

Antipattern a2 can also occur when the *extension* use case would like to inform an actor when it is invoked. The refactoring for this scenario involves explicitly specifying the direction of the association. This guarantees that the use case cannot be initiated by the actor. Unfortunately, UML lacks notation for directed associations. This limitation can be tackled by annotating the association with a UML *comment*.

The rule *RefactorAssociation* in Listing 4 checks for associations between actors and *extension* use cases, and annotates them with a *comment* when found. The *comment* contains the string ‘directed towards’ appended with the name of the actor. Moreover, it swaps the source and destination properties of the association. This is a mere cosmetic

change since the modeler cannot see its effect on the use case diagram. However, this source-destination swap will be reflected in the use case model's XMI file.

```

rule RefactorAssociation {
  from s: UML!Association(
    s.isSourceActor() and s.isDestinationUseCase() and s.destination().isExtension()
  )
  to t: UML!Association (
    memberEnd <- s.memberEnd,
    navigableOwnedEnd <- dst,
    ownedEnd <- Sequence{src, dst}
  ),
  src: UML!Property (
    name <- 'src',
    type <- s.destination()
  ),
  dst : UML!Property (
    name <- 'dst',
    type <- s.source()
  )
  do {
    thisModule.AddComment(s);
    t;
  }
}

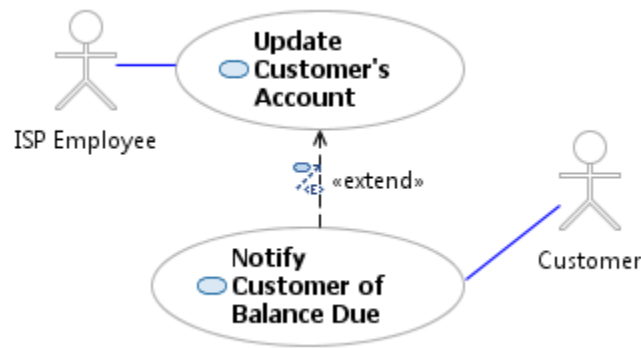
rule CreateComment(a: UML!Association) {
  to t: UML!Comment (
    annotatedElement <- a,
    body <- 'directed towards ' + a.source().name
  )
  do {
    t;
  }
}

```

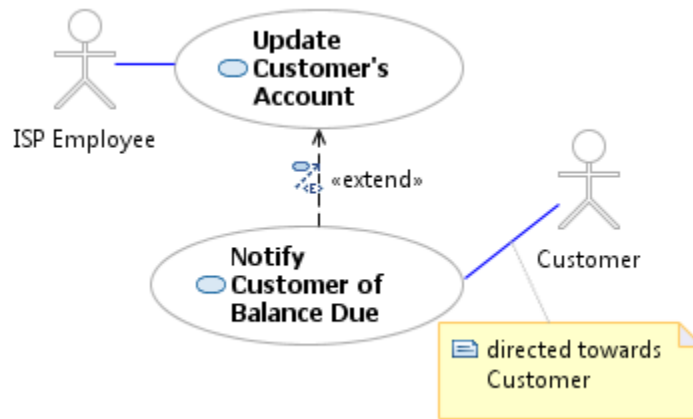
Listing 4: ATL rules for applying Directed Actor-Extension UC Association refactoring

Figure 8 illustrates an example of the *Directed Actor-Extension UC Association* refactoring on a use case model of an Internet Service Provider (ISP) system. In the original use case model (Figure 8(a)), *Notify Customer of Balance Due* is an *extension* use case that executes when an ISP employee would like to inform a customer when his payment is due. Actor *Customer* is associated with this *extension* use case. This association is correct since the *extension* use case informs *Customer* when it is invoked. However, it could also imply that *Customer* can invoke the *extension* use case; this is

incorrect. In order to avoid this incorrect interpretation, the *Directed Actor-Extension UC Association* refactoring is applied on the original use case model. The association between *Customer* and the *extension* use case is annotated with its actual direction, i.e. towards *Customer*. The refactored use case model of the ISP system is shown in Figure 8(b).



(a) Original UC model



(b) Refactored UC model

Figure 8: Example of Directed Actor-Extension UC Association refactoring

a3. Using *extension/inclusion* use cases to implement an *abstract* use case

This antipattern occurs when an *extension* or *inclusion* use case is used to implement an *abstract* use case. The *extension* and *inclusion* use cases describe behavior different from the *abstract* use case. Therefore, the modeler does not use *generalization* relationship to implement the *abstract* use case. A service request from an actor to such an *abstract* use case will never be performed because no use case realizes its intended behavior. Hence, it is inappropriate to model such a relationship between *abstract* and *concrete* use cases. This situation is acceptable if the use case model is incomplete, and the *abstract* use case will be realized later by a *concrete* use case.

r5. Abstract Extended UC to Concrete

This refactoring sets the *abstract* use case to *concrete*. This ensures that the use case can be solely performed, and its intended behavior is realized by itself. A use case is involved in this antipattern if it is:

- *abstract* and associated with at least one actor
- neither a *generalization* nor a *specialization* of any use case
- neither *included* by any use case nor an *extension* of any use case
- *includes* zero or more use cases
- *extended* by at least one use case

The rule *AbstractToConcrete* in Listing 5 checks the above detection conditions for each use case in a given use case model. If a use case satisfies all the detection conditions, its *isAbstract* property is unset.

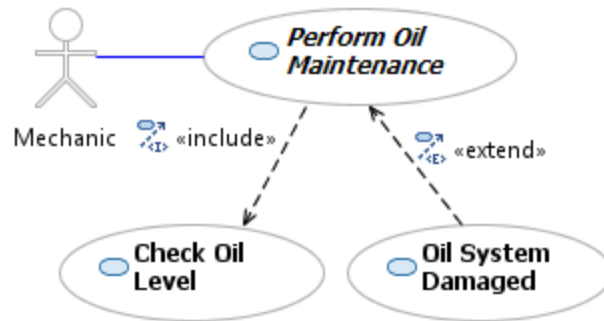
```

rule AbstractToConcrete {
  from s : UML!UseCase (
    s.isAbstract and s.isAssociatedWithActor() and not (s.isIncluded() or
    s.isExtension() or s.isGeneralization() or s.isSpecialization()) and s.isExtended()
  )
  to t: UML!UseCase (
    isAbstract <- false
  )
}

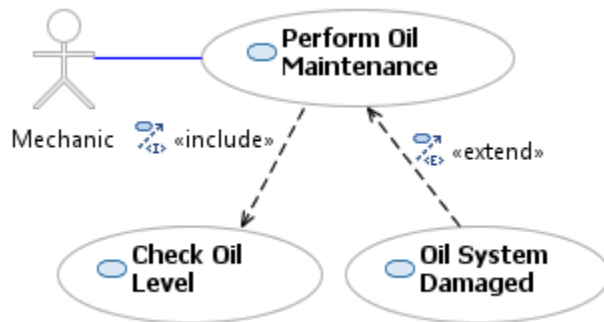
```

Listing 5: ATL rule for applying Abstract Extended UC to Concrete refactoring

Figure 9 illustrates an example of the *Abstract Extended UC to Concrete* refactoring on a use case model of a vehicle repair system. In the original use case model (Figure 9(a)), *Perform Oil Maintenance* is an *abstract* use case which is implemented by *inclusion* use case *Check Oil Level*, and *extension* use case *Oil System Damaged*. Actor *Mechanic* is associated the abstract use case. This indicates that the abstract use case can be performed exclusively. Since abstract use cases are incomplete, *Perform Oil Maintenance* cannot provide complete service to *Mechanic*. Therefore, the *Abstract Extended UC to Concrete* refactoring is applied on the original use case model to result in *Perform Oil Maintenance* set to *concrete*. The refactored use case model of the vehicle repair system is shown in Figure 9(b).



(a) Original UC model



(b) Refactored UC model

Figure 9: Example of Abstract Extended UC to Concrete refactoring

r6. Inclusion to Generalization

This refactoring applies in the case when *inclusion* use cases are used to describe *specialized* behavior of the *abstract* use case. The *inclusion* relationships are replaced by *generalization* relationships directed from the *inclusion* use cases to the *abstract* use case.

An *include* relationship is involved in this antipattern if its *includer* user case is:

- *abstract* and associated with at least one actor
- neither a *generalization* nor a *specialization* of any use case
- neither *included* by any use case nor an *extension* of any use case
- *includes* at least one use case

- not *extended* by any use case

Detection conditions for this refactoring are similar to those of refactoring r5 except for they apply on *include* relationships in a use case model, and the *abstract* use case must *include* at least one use case and should not have any *extensions*. The rule *DropInclude* in Listing 6 checks the above detection conditions for each *include* relationship in a given use case model. If an *include* relationship satisfies all the detection conditions, it is deleted from the use case model. The call to rule *AddGeneralization* is used for introducing *generalization* relationships from the *specializing* use cases to the *abstract* use case.

```

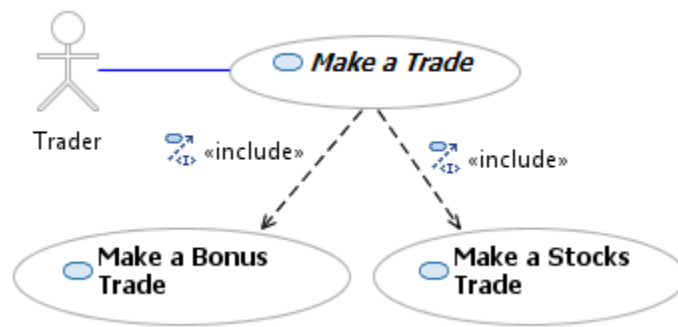
rule DropInclude {
  from s: UML!Include (
    s.getIncluder().isAbstract and s.getIncluder().isAssociatedWithActor() and
    not (s.getIncluder().isIncluded() or s.getIncluder().isExtension() or
    s.getIncluder().isExtended() or s.getIncluder().isGeneralization() or
    s.getIncluder().isSpecialization())
  )
  to drop
  do {
    thisModule.AddGeneralization(s);
  }
}

```

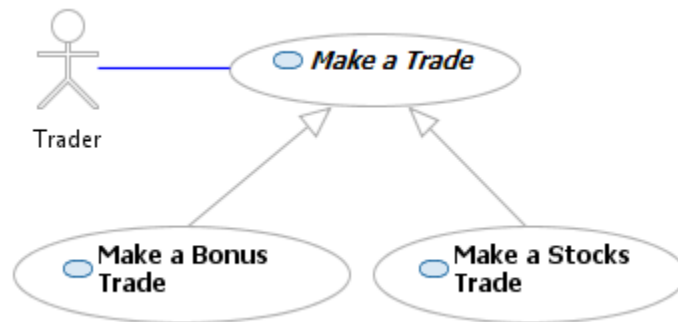
Listing 6: ATL rule for applying Inclusion to Generalization refactoring

Figure 10 illustrates an example of the *Inclusion to Generalization* refactoring on a use case model of a stock market system. In the original use case model (Figure 10(a)), *Make a Trade* is an *abstract* use case which is implemented by *inclusion* uses cases *Make a Bonus Trade* and *Make a Stocks Trade*. Actor *Trader* can perform either of the *inclusion* uses cases. However, *Trader* cannot perform both *inclusion* use cases at the same time. Therefore, the *inclusion* relationships shown in the original use case model

are incorrect. The application of the *Inclusion to Generalization* refactoring on the original use case model replaces the *inclusion* relationships with *generalization* relationships. The new *generalization* relationships are directed from uses cases, *Make a Bonus Trade* and *Make a Stocks Trade*, to the *abstract* use case, *Make a Trade*. This will ensure that a trader can either make a bonus trade, or a stocks trade, but not both at the same time. The refactored use case model of the stock market system is shown in Figure 10(b).



(a) Original UC model



(a) Refactored UC model

Figure 10: Example of Inclusion to Generalization refactoring

a4. Functional Decomposition: Using the *include* relationship

This antipattern represents improper usage of the *include* relationship. The service offered by a use case is divided into several *inclusion* use cases. Moreover, these *inclusion* use cases are not directly associated with any actor. They do not represent a complete service that is offered the system; hence, provide no observable result to a user. Functional decomposition is acceptable if an *inclusion* use case provides complete behavior to another actor and/or is *included* by another use case.

r7. Drop Functional Decomposition

This refactoring merges the *inclusion* use cases into the *base* use case, which individually provides a complete service to the actor. A use case is involved in this antipattern if its:

- an *inclusion* use case which is *included* by one use case only
- not associated with any actor
- neither *including* nor *extending* any use case
- not *extended* by any use case
- neither a *generalization* nor a *specialization* of any use case

The rule *DropUseCase* in Listing 7 checks for *inclusion* use cases in a use case model and deletes them when found. Figure 11 illustrates an example of the *Drop Functional Decomposition* refactoring.

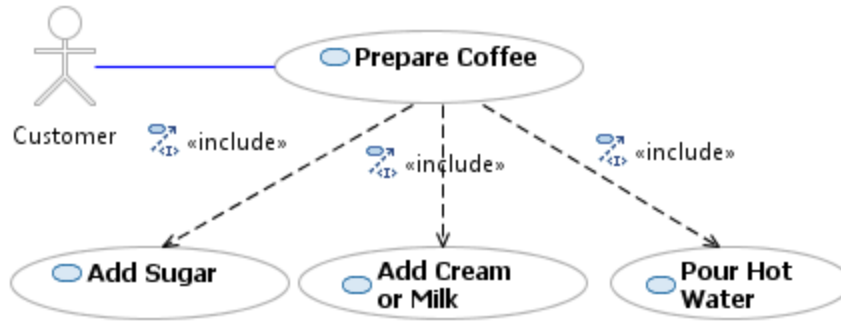
```

rule DropUseCase {
  from s: UML!UseCase (
    not (s.isAssociated() or s.isExtension() or s.isExtended() or s.isGeneralization() or
    s.isSpecialization() or s.isIncluder()) and s.isIncluded() and
    s.getIncluders()->size() = 1
  )
  to drop
}

```

Listing 7: ATL rule for applying Drop Functional Decomposition refactoring

Figure 11 illustrates an example of the *Drop Functional Decomposition* refactoring on a use case model of a coffee vending system. In the original use case model (Figure 11(a)), use case *Prepare Coffee* includes three uses cases *Add Sugar*, *Add Cream or Milk*, and *Pour Hot Water*. All of the *inclusion* use cases are performed when actor *Customer* orders coffee. However, the *inclusion* use cases are neither associated with any actor nor related to any use case, other than *Prepare Coffee*. The *inclusion* use cases actually represent functions of their *base* use case *Prepare Coffee*. Since each of the *inclusion* use cases do not represent a complete service provided by the system, the *Drop Functional Decomposition* refactoring is applied on the original use case model. The *inclusion* use cases are removed from the use case model, and their behavior is implicitly merged into the *base* use case. Figure 11(b) shows the refactored use case model of the coffee vending system.



(a) Original UC model



(b) Refactored UC model

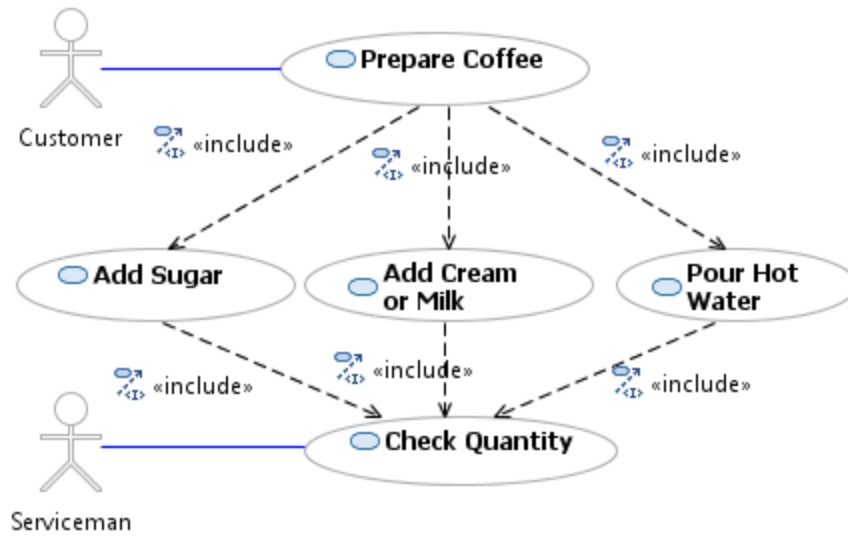
Figure 11: Example of Drop Functional Decomposition refactoring

r8. Drop Functional Decomposition having *Inclusion*

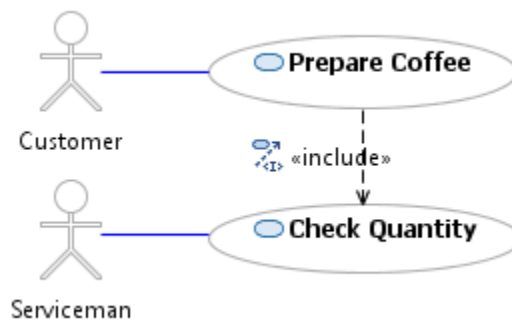
Functional Decomposition can also occur if the *inclusion* use case *includes* other use case(s). In this case, the refactoring has an additional step. After the *inclusion* use case's deletion, *include* relationships are added from its *base* uses case to its *inclusion* use cases.

Figure 12 illustrates an example of the *Drop Functional Decomposition having Inclusion* refactoring on an enhanced use case model of the coffee vending system, described in refactoring r7. In the original use case model (Figure 12(a)), the *inclusion* use cases *Add Sugar*, *Add Cream or Milk*, and *Pour Hot Water* further *include* use case *Check Quantity*. The inclusion use cases check the quantity of sugar, cream, milk, and water, by invoking *Check Quantity*. Actor *Serviceman* also performs *Check Quantity*

while maintaining the coffee machine. As explained in refactoring r7, each of the *inclusion* use cases do not represent a complete service provided by the system. Since they *include* a use case, *Check Quantity*, the *Drop Functional Decomposition having Inclusion* refactoring is applied on the original use case model. The *inclusion* use cases are removed from the use case model, and their behavior is implicitly merged into the *base* use case. Moreover, an *include* relationship is added from *Prepare Coffee* to *Check Quantity*. This additional step is performed in order to preserve the behavior shown in the original use case model. Figure 12 (b) shows the refactored use case model of the coffee vending system.



(a) Original UC model



(b) Refactored UC model

Figure 12: Example of Drop Functional Decomposition having Inclusion refactoring

a5. Functional Decomposition: Using the *extend* relationship

This antipattern represents improper usage of the *extend* relationship, in which a single use case extends multiple *base* use cases. To elaborate, the *extension* use case is providing optional behavior which is useful to multiple *base* use cases. This strongly indicates that the *extension* use case has degraded into a function, and cannot properly provide optional behavior to its *base* use cases.

r9. Split Extension UC

This refactoring splits the *extension* use case into multiple use cases, each of which provide optional behavior specific to a single *base* use case. This will ensure that exceptional situations are properly handled by the *extension* use cases. A use case is involved in this antipattern if it is:

- not associated with any actor
- neither a *generalization* nor a *specialization* of any use case
- neither an *inclusion* use case nor *including* any use case
- not *extended* by any use case
- *extends* more than one use case

The rule *DropUseCase* in Listing 8 checks for *extension* use cases that are shared by multiple *base* use cases, and deletes them when found. The call to rule *AddUseCase* adds specific *extension* use cases into the use case model for each *base* use case. The name of the *extension* use case is appended with the name of its *base* use case followed by ‘Extension’ in parenthesis. This indicates the modeler to rename this use case appropriately.

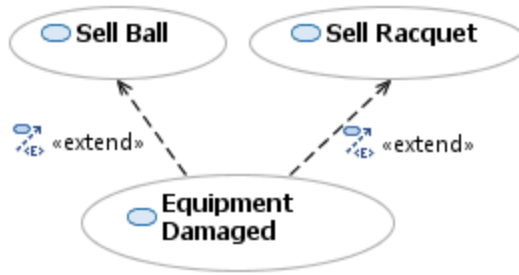
```

rule DropUseCase {
  from s: UML!UseCase (
    not(s.isAssociated() or s.isIncluded() or s.isIncluder() or s.isExtended()
    or s.isGeneralization() or s.isSpecialization()) and s.extend->size() > 1;
  )
  to drop
  do {
    for(ex in s.extend) {
      thisModule.AddUseCase(ex);
    }
  }
}

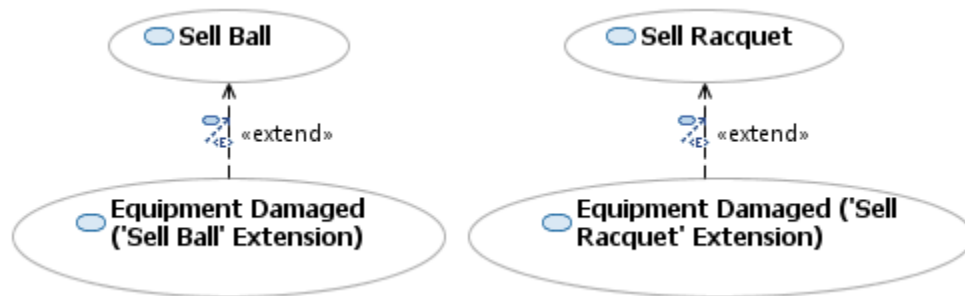
```

Listing 8: ATL for applying Split Extension UC refactoring

Figure 13 illustrates an example of the *Split Extension UC* refactoring on a use case model of a sports store system. In the original use case model (Figure 13(a)), *Equipment Damaged* is an *extension* use case of two *base* use cases, *Sell Ball* and *Sell Racquet*. The *extension* use case gets invoked when damaged merchandise, either ball or racquet, is being sold. When the *extension* use case is invoked by *Sell Ball*, additional functionality will be performed for handling the damaged ball. However, redundant functionality will be performed for handling a damaged racquet, which is not being sold. Similarly, when the *extension* use case is invoked by *Sell Racquet*, redundant functionality is performed for handling a damaged ball. In order to avoid this redundant functionality, the *Split Extension UC* refactoring is applied on the original use case model. The *extension* use case is split into new two use cases, each of which provide required optional behavior to their respective *base* use cases. The new use cases must be renamed appropriately by the modeler. Figure 13(b) shows the refactored use case model of the sports store system.



(a) Original UC model



(b) Refactored UC model

Figure 13: Example of Spilt Extension UC refactoring

r10. Extension to Generalization

This refactoring is applied in case the *extension* use case is used to depict specialized behavior of the *base* use case. The *extend* relationship is replaced with an appropriate *generalization* relationship. An *extend* relationship is involved in this antipattern if its *extension* use case is:

- not associated with any actor
- neither a *generalization* nor a *specialization* of any use case
- neither an *inclusion* use case nor *including* any use case
- not *extended* by any use case
- *extends* more than one use case

Detection conditions for this refactoring are similar to those of refactoring r9 except for they apply on *extend* relationships in a use case model. The rule *DropExtend* in Listing 9 checks for *extend* relationships whose *extension* use case is shared by multiple *base* use cases, and deletes them when found. The call to rule *AddGeneralization* adds a *generalization* relationship from the *extension* use case to its respective *base* use case.

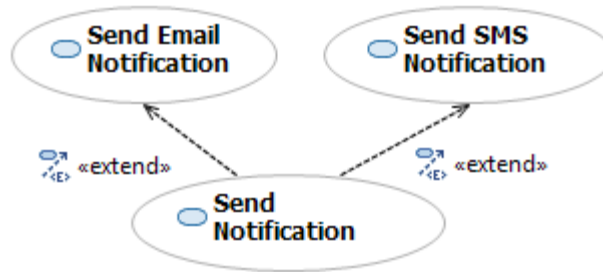
```

rule DropExtend {
  from s: UML!Extend (
    not (s.getExtension().isAssociated() or s.getExtension().isIncluded() or
    s.getExtension().isIncluder() or s.getExtension().isExtended() or
    s.getExtension().isGeneralization() or s.getExtension().isSpecialization()) and
    s.getExtension().extend->size() > 1
  )
  to drop
  do {
    thisModule.AddGeneralization(s);
  }
}

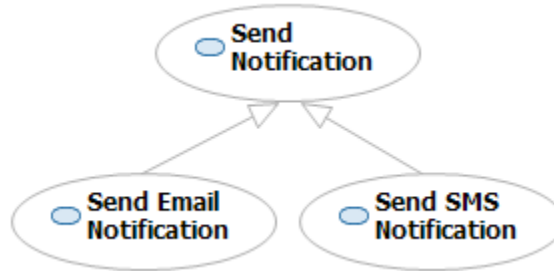
```

Listing 9: ATL rule for applying Extension to Generalization refactoring

Figure 14 illustrates an example of the *Extension to Generalization* refactoring on a use case model of a notification system. In the original use case model (Figure 14 (a)), use case *Send Notification* extends two base use cases *Send Email Notification* and *Send SMS Notification*. The *extend* relationships are used to represent the hierarchy of notification services offered by the system. This hierarchy of services is correctly represented by *generalization* relationships. Therefore, the *Extension to Generalization* refactoring is applied on the original use case model. Figure 14 (b) shows the refactored use case model of the notification system.



(a) Original UC model



(b) Refactored UC model

Figure 14: Example of Extension to Generalization refactoring

a6. Multiple *generalizations* of a use case

This antipattern occurs when a single use case *implements* more than one use case. Common behavior from the *base* use cases is extracted and represented in a *specialized* use case.

r11. Generalization to Inclusion

A use case should not *specialize* multiple *base* use cases at the same time. This strongly indicates that behavioral semantics of the model are violated, and leads to incorrect implementation of the system. This refactoring replaces *generalization* relationships with *include* relationships directed from the *generalized* use cases to the *specialized* use case.

The rule *DropGeneralization* in Listing 10 deletes a *generalization* relationship if its source use case has multiple *generalizations*. The call to rule *CreateInclude* introduces *include* relationships between *generalized* and *specialized* use cases.

```

rule DropGeneralization {
  from s: UML!Generalization (
    s.refImmediateComposite().hasMultipleGeneralizations()
  )
  to drop
}

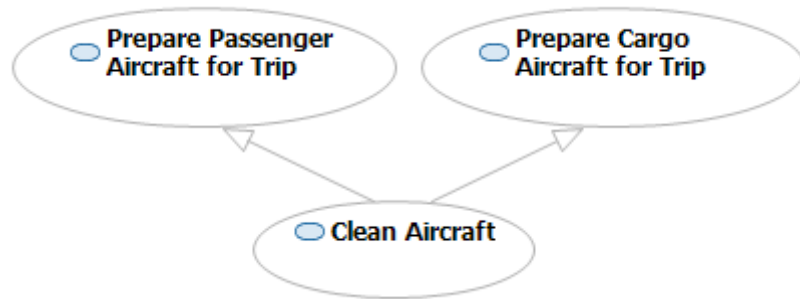
rule CopyUseCase {
  from s : UML!UseCase
  to t: UML!UseCase (
    name <- s.name,
    include <- s.include,
    extend <- s.extend,
    generalization <- s.generalization,
    isAbstract <- s.isAbstract
  )
  do {
    for(uc in s.getSpecializations()) {
      if(uc.hasMultipleGeneralizations()) {
        t.include <- t.include->including(thisModule.CreateInclude(s, uc));
      }
    }
  }
  t;
}

```

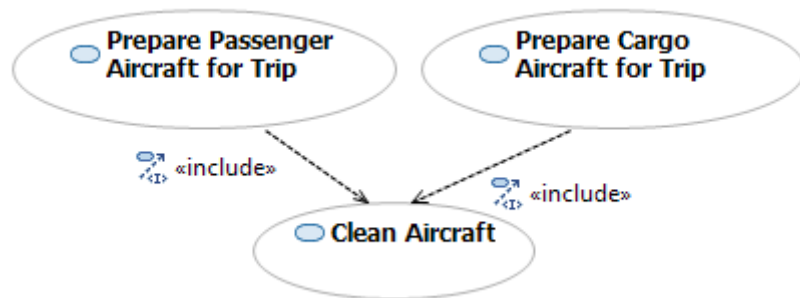
Listing 10: ATL rule for applying Generalization to Inclusion refactoring

Figure 15 illustrates an example of the *Generalization to Inclusion* refactoring on a use case model of an aircraft management system. In the original use case model (Figure 15(a)), use case *Clean Aircraft* inherits behavior from two *generalized* use cases, *Prepare Passenger Aircraft for Trip* and *Prepare Cargo Aircraft for Trip*. The modeler intended to extract the common behavior of the generalized use cases into *Clean Aircraft*. This implies that in order to clean an aircraft, a passenger aircraft, and a cargo aircraft must be prepared. This is incorrect behavior of *Clean Aircraft* because only one aircraft can be cleaned at a time. Therefore, the *Generalization to Inclusion* refactoring is applied on the original use case model. The refactoring replaces the incorrect *generalization*

relationships with *inclusion* relationships. This will ensure that proper behavior is performed when *Clean Aircraft* is invoked. Figure 15(b) shows the refactored use case model of the aircraft management system.



(a) Original UC model



(a) Refactored UC model

Figure 15: Example of Generalization to Inclusion refactoring

a7. Use cases containing common and exceptional functionality

This antipattern occurs when a use case is reused by making it an *inclusion* and *extension* for different *base* use cases. This shared use case contains common and optional behavior required by multiple use cases. When the shared use case is initiated by any of the *base* use cases, extra undesired functionality is performed.

r12. Drop Inclusion

If the shared use case represents functionality which is appropriate only for the *base* use case it *extends*, its *inclusion* relationship should be deleted. A new *inclusion* use case is added in order to provide the additional behavior required by the other *base* use case.

The ATL helper attribute *sharedUCs* in Listing 11 contains the set of use cases which are *inclusions* and *extensions*. The rule *DropInclude* deletes an *inclusion* relationship if its target use case is a shared use case. The call to rule *AddUseCase* adds a new *inclusion* use case to the use case model. The name of the *base* use case is copied into this new use case and appended by ‘Inclusion’ in parenthesis. This indicates the modeler to rename this use case appropriately.

```
helper def: sharedUCs : Set(UML!UseCase)
= UML!Include->allInstances()
  ->collect(uc | uc.addition->asSet()
  ->select(uc | uc.extend->size() > 0);

rule DropInclude {
  from s: UML!Include (
    thisModule.sharedUCs->includes(s.addition)
  )
  to drop
  do {
    thisModule.AddUseCase(s);
  }
}
```

Listing 11: ATL rule for applying Drop Inclusion refactoring

r13. Drop Extension

If the shared use case represents functionality appropriate only for the *base* use case that *includes* it, its *extension* relationship should be deleted. A new *extension* use case is

added in order to provide the optional behavior required by the other (*extended*) *base* use case.

The rule *DropExtend* in Listing 12 deletes an *extend* relationship if its source use case is a shared use case. The call to rule *AddUseCase* adds a new *extension* use case to the use case model. The name of the *base* use case is copied into this new use case and appended by ‘Extension’ in parenthesis. This indicates the modeler to rename this use case appropriately.

```
rule DropExtend {
  from s: UML!Extend (
    thisModule.sharedUCs->includes(s.getExtension())
  )
  to drop
  do {
    thisModule.AddUseCase(s);
  }
}
```

Listing 12: ATL rule for applying Drop Extension refactoring

In case the shared use case does indeed contain both additional and optional behavior required for the *base* use cases, it must be split into two separate use cases. The new use cases provide appropriate functionality to their respective *base* use cases.

Figure 16 illustrates an example of the *Drop Inclusion* and *Drop Extension* refactoring on a use case model of a car dealership system. In the original use case model (Figure 16(a)), use case *Car Not Found* is *included* by use case *Add New Car*, and *extends* use case *Update Car’s Information*. When a new car is added into the system, the actor *Car Salesman* performs *Add New Car*. Before a car is added into the system, the system must check whether it is already available or not. This is represented by the

behavior in the shared use case *Car Not Found*. The *Car Salesman* can change information related to a car by performing *Update Car's Information*, which requires the car's unique identifier. If the car is not found in the system, the shared use case is performed to generate an error report. The shared use case behaves differently when invoked by each of its *base* use cases. This means that redundant functionality will be performed when either of the *base* use cases invoke the shared use case. Therefore, the *Drop Inclusion* and *Drop Extension* refactoring are applied on the original use case model. The refactorings split the shared use case into two use cases, each of which provide required behavior to their respective base use cases. The new use cases must be renamed appropriately by the modeler. Figure 16(b) shows the refactored use case model of the car dealership system.

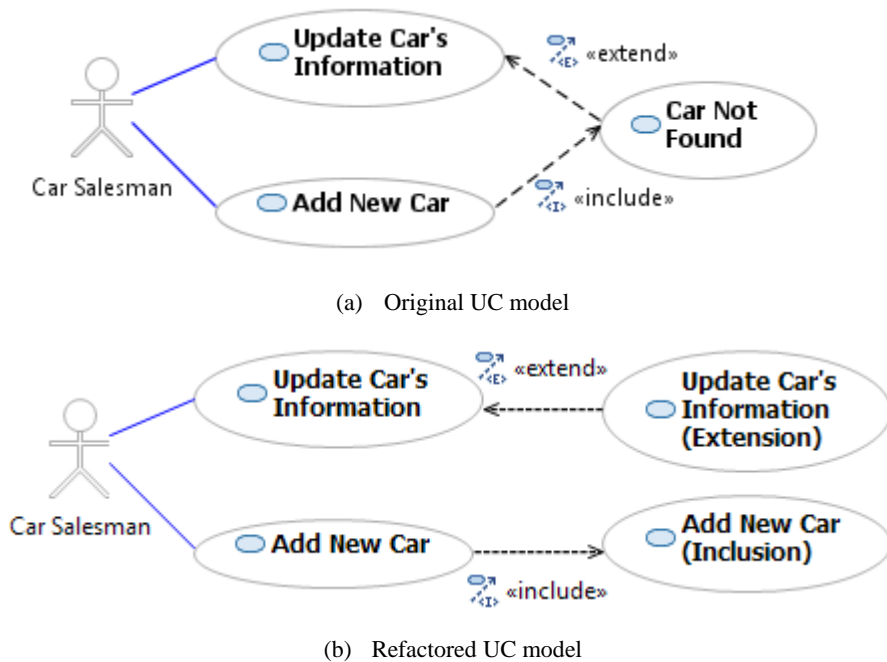


Figure 16: Example of Drop Inclusion and Drop Extension refactorings

a8. Multiple actors associated with one use case

This antipattern occurs when a use case is associated with more than one actor. The actors perform the same role while interacting with the shared use case. This association is inappropriate since it is against the semantics of an actor. An actor must perform a unique role while interacting with a shared use case. This eventually results in multiple implementations of the shared use case for different actors. In case the shared use case needs to communicate with distinct actors, this situation is acceptable.

r14. Generalize Actors

This refactoring deletes the associations between the actors and shared use case, introduces a *generalized* actor, and associates the shared use case with it. The *generalized* actor represents the similar roles performed by actors while executing the shared use case.

A pair of actors is involved in this antipattern if they are associated with at least one common use case. The call to rule *AddGeneralizedActor* in Listing 13 adds a generalized actor to the use case model and associates it with the common use case(s). This actor is named 'Super Actor'; the modeler must rename it appropriately.

```

rule CopyPackage {
  from s: UML!Package
  to t: UML!Package (
    name <- s.name,
    packagedElement <- s.packagedElement,
    ownedComment <- s.ownedComment
  )
  do {
    for(saps in thisModule.sharedActorPairs) {
      thisModule.AddGeneralizedActor(saps, t);
    }
  }
}

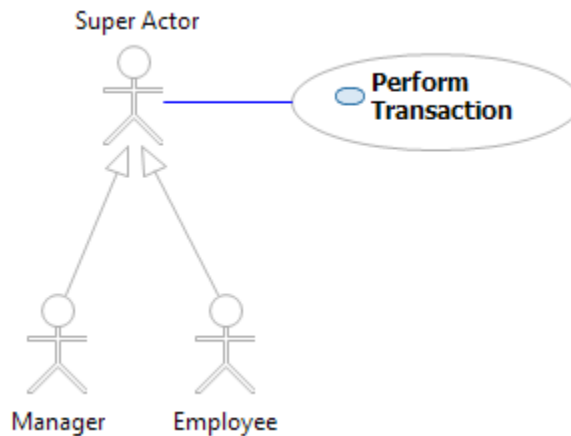
```

Listing 13: ATL rule for applying Generalize Actor refactoring

Figure 17 illustrates an example of the *Generalize Actors* refactoring on the use case model of a banking system. In the original use case model (Figure 17(a)), actor *Manager* and *Employee* are associated with the same use case *Perform Transaction*. The actors associated play a similar role when performing the shared use case. In other words, the actors will communicate with the shared use case in a similar fashion. For example, the procedure for performing *Perform Transaction* is the same when performed by *Manager* or *Employee*. Actors should communicate with a use case if they are playing unique roles while the use case is being performed. Therefore, in the original use case model, designers will assume that *Manager* and *Employee* play different roles when executing *Perform Transaction*. Hence, the implementation of the actors with respect to the execution of the use case will be different, even though they should be the same. This scenario can be fixed by applying the *Generalize Actors* refactoring on the original use case model. The refactoring extracts the overlapping roles between the associated actors, and creates a new actor, *Super Actor*, that represents these roles. The involved actors will generalize the newly created actor. Figure 17(b) shows the refactored use case model of the banking system.



(a) Original UC model



(b) Refactored UC model

Figure 17: Example of Generalize Actors refactoring

r15. Spilt UCs

This refactoring must be applied in case the functionality offered by the shared use case is too generic for servicing the requests of both actors. The refactoring splits the shared use case into appropriate use cases for each actor. These new use cases accurately depict the intended behavior of the system when interacting with each actor.

The rule *DropUseCase* in Listing 14 checks for shared use cases in a use case model and deletes them when found. The call to rule *AddUseCase* adds new use cases into the use case model for each of its associated actors. The name of the shared use case

is copied into the new use cases and appended with its respective actor's name in parenthesis. This indicates the modeler to rename this use case appropriately.

```
rule DropUseCase {
  from s : UML!UseCase (
    thisModule.ucsToBeDroppend->includes(s)
  )
  to drop
  do {
    for(ac in s.getAssociatedActors()) {
      thisModule.AddUseCase(s, ac);
    }
  }
}
```

Listing 14: ATL rule for applying Split UCs refactoring

The refactorings for this antipattern have been implemented for pairs of actor that share common use case(s). They can be extended for a larger set of actors.

a9. An association between two actors

This antipattern occurs when an association relationship between two actors is shown in a use case model. Association between actors represents interactions that are external to the system. A use case model should be concerned only with the interactions between a system and its actors. Incorporating interactions between external entities adds unnecessary complexity to a use case model.

r16. Drop Actor-Actor Association

This refactoring deletes an association between a pair of actors. This will ensure that modelers focus on interactions between a system and its actors, rather than external interactions.

A given association relationship in a use case model is involved in this antipattern if its source and destination are both actors. The rule *DropAssociation* in Listing 15 checks for an association whose source and destination are actors, and deletes them when found.

```
rule DropAssociation {
  from s: UML!Association (
    s.isSourceActor() and s.isDestinationActor()
  )
  to drop
}
```

Listing 15: ATL rule for applying Drop Actor-Actor Association refactoring

a10. An association between use cases

This antipattern occurs when a use case model contains an association between a pair of use cases. This association relationship represents communication between the use cases in order to provide complete service to an actor.

r17. Drop UC-UC association

A use case model must be concerned with interactions between a system and its actors. Representing internal interactions adds unnecessary complexities into the use case model. This refactoring merges a pair of associated use cases into a single use case, which provides a complete and meaningful functionality to system users. Association relationships between actors and the associated use cases are directed towards the merged use case.

The rule *DropAssociation* in Listing 16 deletes associations between pairs of use cases. The call to rule *AddUseCase* adds a new use case into the use case model. This use case represents combined functionality of the associated use case. This use case is named ‘Merged UC’ followed by names of the associated use cases in parenthesis. This indicates the modeler to rename the use case appropriately.

```
rule DropAssociation {
  from s: UML!Association (
    s.isSourceUseCase() and s.isDestinationUseCase()
  )
  to drop
  do {
    thisModule.AddUseCase(s);
  }
}
```

Listing 16: ATL rule for applying Drop UC-UC Association refactoring

Figure 18 illustrates an example of the *Drop UC-UC Association* refactoring on a use case model of a vehicle embedded system. In the source use case model (Figure 18(a)), two use cases *Count Shaft Rotations in Trip* and *Measure Time of Trip* are associated. The former is responsible for calculating the distance traveled during a trip, whereas the latter tracks the time spent during a trip. These use cases, by themselves, do not provide any service to the user. They provide the necessary information required for calculating the average speed of the car during a trip. Therefore, the *Drop UC-UC Association* refactoring is applied on the original use case model. The refactoring merges the two use cases into a single use case, which calculates the average speed of the car during a trip. The merged use case should be renamed properly by the modeler. Figure 18 (b) shows the refactored use case model of the car dealership system.



(a) Original UC model



(b) Refactored UC model

Figure 18: Example of Drop UC-UC Association refactoring

a11. An unassociated use case

This antipattern occurs when a use case model contains a use case that is not associated with any actor. Such a use case represents functionality that is internal to the system; therefore, does not provide any service to the system's user. This situation is acceptable if the use case model is incomplete.

r18. Drop Unassociated UC

This refactoring deletes an unassociated use case from the use case model. The purpose of use case modeling is to model the interactions between a system and its actors. Hence, internal functionality should not be represented in a use case model.

The rule *DropUseCase* in Listing 17 checks for unassociated use cases in a use case model, and deletes them when found. Apart from being not associated with any actor, the use case must not be involved in any *inclusion*, *extension* and *generalization* relationship.

```
rule DropUseCase {
  from s : UML!UseCase (
    not (s.isAssociated() or s.isExtension() or s.isExtended() or s.isIncluder()
    or s.isIncluded() or s.isGeneralization() or s.isSpecialization())
  )
  to drop
}
```

Listing 17: ATL rule for applying Drop Unassociated UC refactoring

a12. Two actors with same name

This antipattern occurs when several actors in the same use case model have identical names. This situation may occur if an actors' roles is carried out by different personnel with similar job titles. This situation is acceptable if several instances of an actor can enhance the layout of a use case diagram.

r19. Rename Actor

Actors with identical names are a source of confusion in a use case model. The identical actors should be renamed such that their responsibilities can be distinguished, and represented more precisely.

The rule *RefactorActor* in Listing 18 checks for a duplicate actor and renames it when found. The actor is renamed 'Duplicate Actor' followed by the actor's original name in parenthesis. This indicates the modeler to consider renaming the actor appropriately.

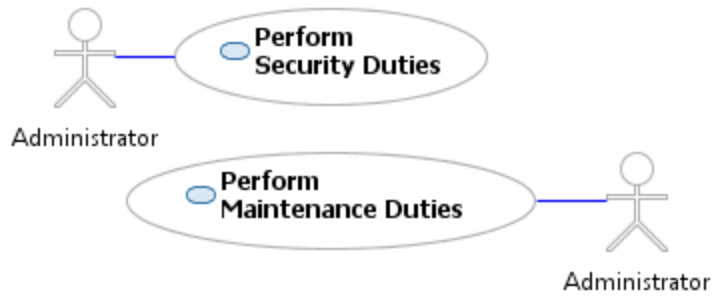
```

rule RefactorActor {
  from s : UML!Actor (
    UML!Actor->allInstances()
    ->excluding(s)
    ->collect(a | a.name)
    ->includes(s.name)
  )
  to t: UML!Actor (
    name <- 'Duplicate Actor ' + '(' + s.name + ')'
  )
}

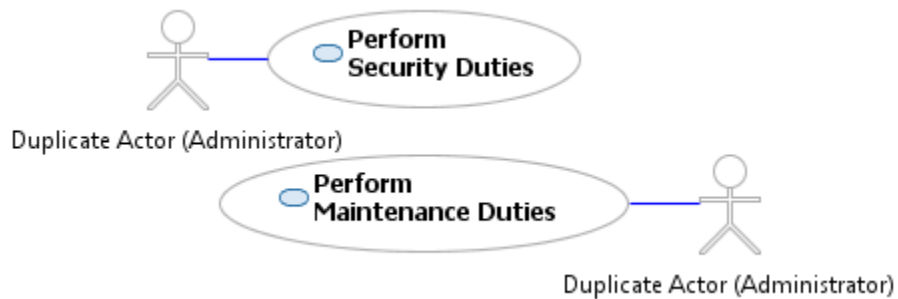
```

Listing 18: ATL rule for applying Rename Actor refactoring

Figure 19 illustrates an example of the *Rename Actor* refactoring. In the original use case model (Figure 19(a)), two actors have an identical name, *Administrator*. This actor's role is performed by different personnel with identical job titles. The first type of role is security administration, and the second one is maintenance administration. Therefore, the *Rename Actor* refactoring is applied on the original use case model. The refactoring renames the identical actors in order to indicate the modeler of duplication. The refactored use case model is shown in Figure 19(b).



(a) Original UC model



(b) Refactored UC model

Figure 19: Example of Rename Actor refactoring

a13. An actor associated with an unimplemented *abstract* use case

This antipattern occurs when an actor is directly associated with an *abstract* use case that is not implemented by any *specialized* use case(s). A service request from an actor to such a use case will not be performed since the use case cannot be initiated. This situation is acceptable if the use case model is incomplete. The use case modelers are expected to later add *concrete* use case(s) which will implement the *abstract* use case. However, assuming that the *abstract* use case can be initiated to provide service to an actor is incorrect.

r20. Abstract to Concrete

This refactoring converts the *abstract* use case to *concrete*. This ensures that the use case can be performed and intended service will be provided to the actor. A use case is involved in this antipattern if it is:

- *abstract* and associated with at least one actor
- neither a *generalization* nor a *specialization* of any use case
- neither *including* nor *extending* any use case
- neither *included* nor *extended* by any use case

The rule *AbstractToConcrete* in Listing 19 checks the above detection conditions for each use case in a given use case model. If a use case satisfies all the detection conditions, its *isAbstract* property is unset. Figure 20 illustrates an example of the *Abstract to Concrete* refactoring.

```
rule AbstractToConcrete {
  from s: UML!UseCase (
    s.isAssociatedWithActor() and s.isAbstract and not (s.isGeneralization() or
    s.isSpecialization() or s.isIncluder() or s.isIncluded() or s.isExtension() or
    s.isExtended())
  )
  to t: UML!UseCase (
    isAbstract <- false
  )
}
```

Listing 19: ATL rule for applying Abstract to Concrete refactoring



(a) Original UC model



(b) Refactored UC model

Figure 20: Example of Abstract to Concrete refactoring

r21. Add Concrete UC

This refactoring must be applied in case the *abstract* use case is indeed incomplete. A *concrete* use case that implements the *abstract* use case is added into the use case model. A *generalization* relationship is also added from the *concrete* use case to the *abstract* use case. This guarantees that the *abstract* use case will not be solely performed. Therefore, complete and meaningful service is provided to an actor through the *concrete* use case.

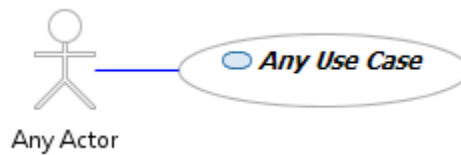
The rule *RefactorUseCase* in Listing 20 detects unimplemented use cases in a given use case model, and refactors them by adding a *concrete* use case which implements them. The name of the *abstract* use case is copied into the *concrete* use case and appended with ‘Concrete’ in parenthesis. This indicates the modeler to rename the *concrete* use case appropriately. Figure 21 illustrates an example of the *Add Concrete UC* refactoring.

```

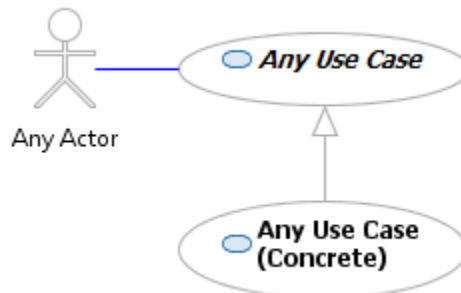
rule RefactorUseCase {
  from s: UML!UseCase (
    s.isAssociatedWithActor() and s.isAbstract and not (s.isGeneralization() or
    s.isSpecialization() or s.isIncluder() or s.isIncluded() or s.isExtension() or
    s.isExtended())
  )
  to t: UML!UseCase (
    name <- s.name,
    isAbstract <- s.isAbstract
  )
  do {
    thisModule.AddSpecializedUseCase(s,t);
  }
}

```

Listing 20: ATL rule for applying Add Concrete UC refactoring



(a) Original UC model



(b) Refactored UC model

Figure 21: Example of Add Concrete UC refactoring

3.2 Case Study

In this section we present a real world case study to demonstrate the feasibility of the proposed approach.

3.2.1 Definition and Motivation

The main research questions posed by this case study are as follow:

R1: Does the model transformation approach have the same antipattern detection capabilities as the detection technique presented in [19]? If a different set of antipattern matches are detected then these matches must be investigated to compare the coverage of the antipattern sets detected. Comparing the coverage of the two sets of antipattern matches will indicate which detection technique is superior.

R2: Does model transformation apply the refactorings correctly? Given a set of antipattern matches and an identified set of required refactorings, the refactorings will be applied using model transformation. The correctness of the refactorings carried out will be initially verified by examining the target models structurally. Correctness will then be verified once again by comparing the target models with manually refactored use case models presented in [19]. Any discrepancy between them will be investigated to decide which approach is superior.

3.2.2 Formulation

In order to address the above mentioned research questions, the use case model that was used in the case study presented in [19] will be reused for comparative purposes. The use case model used in the case study presented in [19] suffered from a set of documented issues. Antipatterns in the use case model were then searched for using ARBIUM. The antipattern matches prompted the execution of a set of corresponding refactorings that were carried out manually, which subsequently resolved the documented

list of issues in the use case model. Therefore, in the first phase of this case study, the proposed approach will be used to detect antipatterns in the same use case model. The two sets of antipattern matches will then be compared. In the second phase of this case study, the manually applied refactorings in [19] will be applied using model transformation. The target models will be examined directly to determine their correctness. Moreover, the target models will be compared with the target models presented in [19]. The results of these evaluations are presented in Section 3.3.

The case study pertains to the use case model of the MAPSTEDI (Mountains and Plains Spatio-Temporal Database Informatics) system [116]. The MAPSTEDI system is a distributed database system that integrates biodiversity data collections from three sources; the University of Colorado Museum (UCOM); the Denver Museum of Nature and Science (DMNS); and the Denver Botanic Gardens (DBG). The integrated database contains 285,000 biological specimens. The system will allow geocoders to analyze biodiversity data in the southern and central Rocky Mountains. A map based GUI is provided by MAPSTEDI to allow users to geographically reference the specimens.

The use case model of MAPSTEDI (Figure 22-Figure 26) consists of five packages, each of which model the functional requirements of individual subsystems. Each package is individually checked for presence of antipattern instances. The use case model is accompanied by textual descriptions for each individual use case. The use case descriptions are required for determining the validity of the use case model.

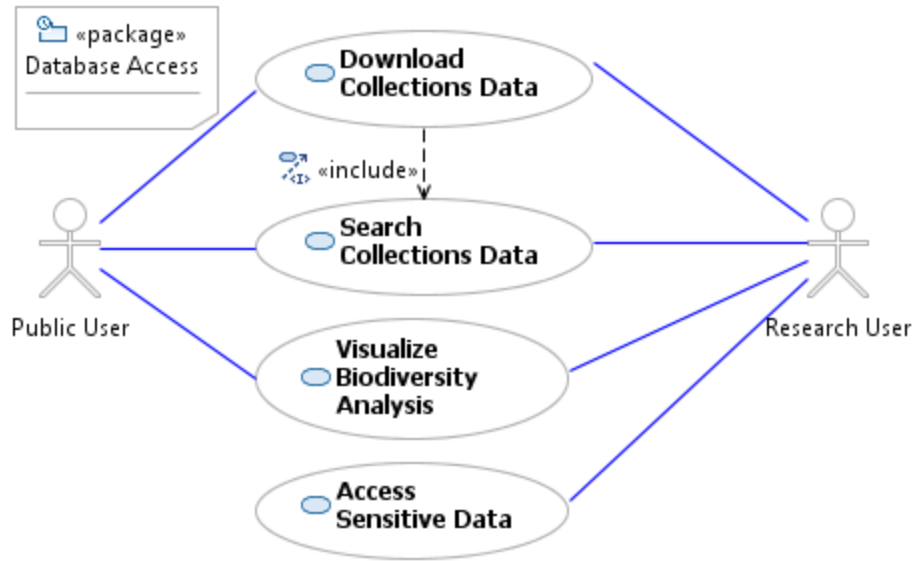


Figure 22: Use case model of Database Access subsystem

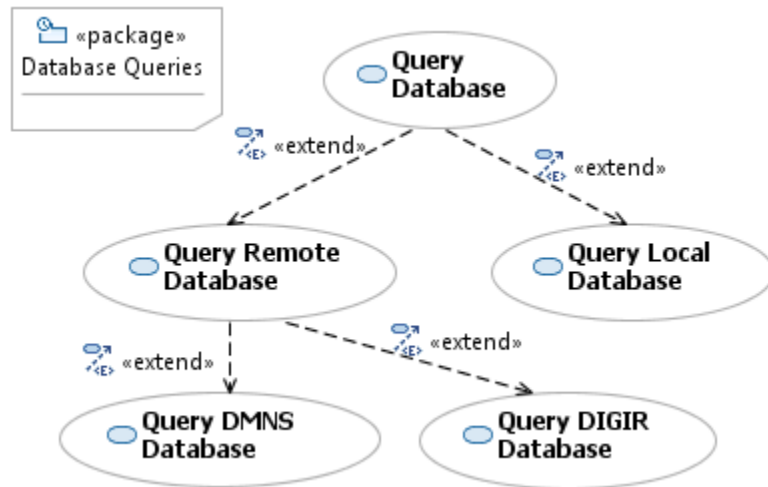


Figure 23: Use case model of Database Queries subsystem

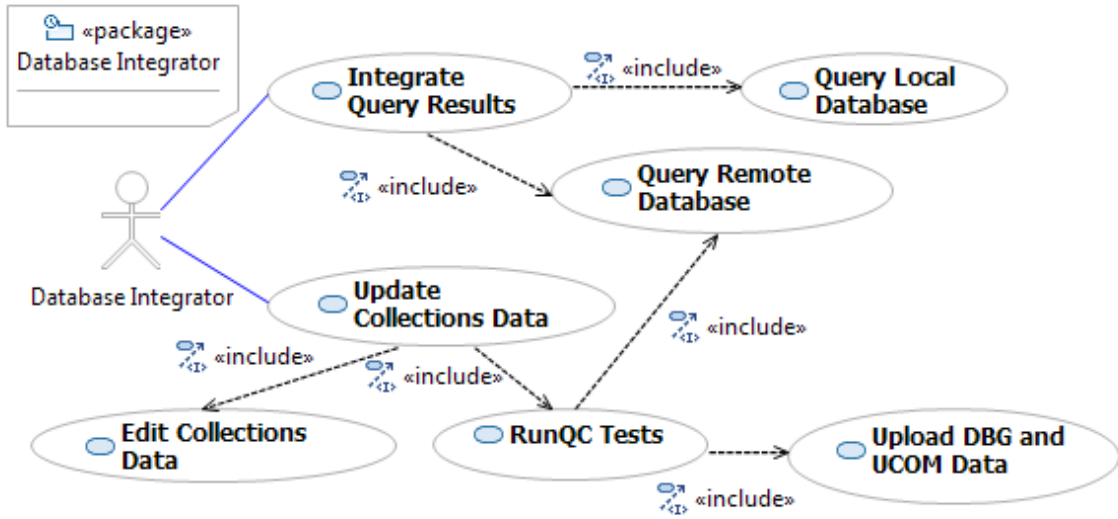


Figure 24: Use case model of Database Integrator subsystem

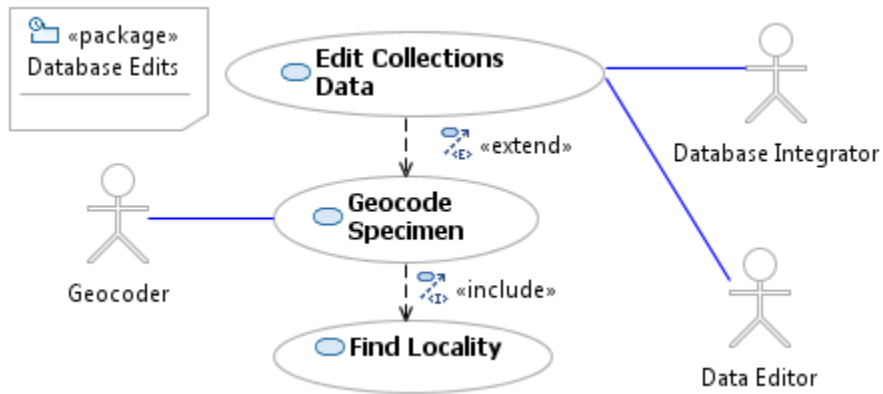


Figure 25: Use case model of Database Edits subsystem

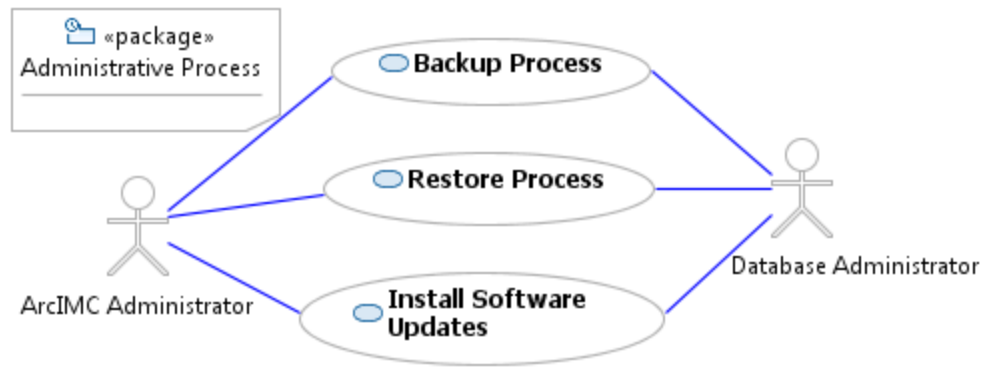


Figure 26: Use case model of Administrative Process subsystem

3.2.3 Model Transformations

This section will describe and illustrate antipattern detection, and refactoring on the use case models of the MAPSTEDI system.

Database Access Subsystem

In the *Database Access* use case model, two actors are associated with a common set of use cases. Actors *Public User* and *Research User* are both associated with use cases *Download Collections Data*, *Search Collection Data* and *Visualize Biodiversity Analysis*. This matches antipattern a8. Now, the question arises which one of the two refactorings r14 and r15 must be applied to this antipattern instance. The answer to this question lies in the use case descriptions of the use cases involved in this antipattern. Analysis of the use case descriptions reveals that actors *Public User* and *Research User* perform similar roles when executing the shared use cases. This suggests that refactoring r14 must be applied on the antipattern instance. The refactoring adds an actor *Super Actor* to the use case model that *generalizes* the similar roles performed by actors *Public User*

and *Research User*. The added actor is associated with the shared use cases *Download Collections Data*, *Search Collection Data* and *Visualize Biodiversity Analysis*. The associations from actors *Public User* and *Research User* to the shared use cases are deleted. The actor *Super Actor* must be renamed appropriately by the modeler. Figure 27 presents the refactored use case model of the *Database Access* subsystem.

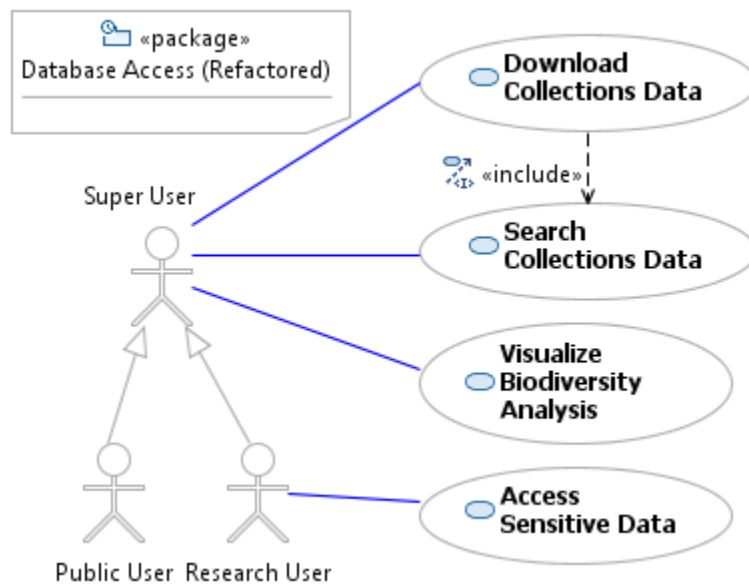


Figure 27: Use case model of Database Access subsystem after applying the Generalize Actors refactoring

Database Queries Subsystem

In the *Database Queries* use case model *extension* use cases, *Query Database* and *Query Remote Database*, each provide optional functionality to more than one *base* use cases. This matches antipattern a5. Two instances of this antipattern exist in the *Database Queries* use case model. In the first instance, use case *Query Database* extends two use

cases, *Query Remote Database* and *Query Local Database*. In the second instance, use case *Query Remote Database* extends *Query DMNS Databases* and *Query DIGIR Database*. The two instances result in a hierarchy of functional decompositions. Analysis of the use case descriptions of these *extension* use cases indicates that this hierarchy is incorrect. The functionalities provided by the *extension* use cases are in fact *specialized* versions of general behavior described by their respective *base* use cases. Therefore, refactoring r10 is applied on the model elements involved in this antipattern. Figure 28 presents the refactored use case model of the *Database Queries* subsystem.

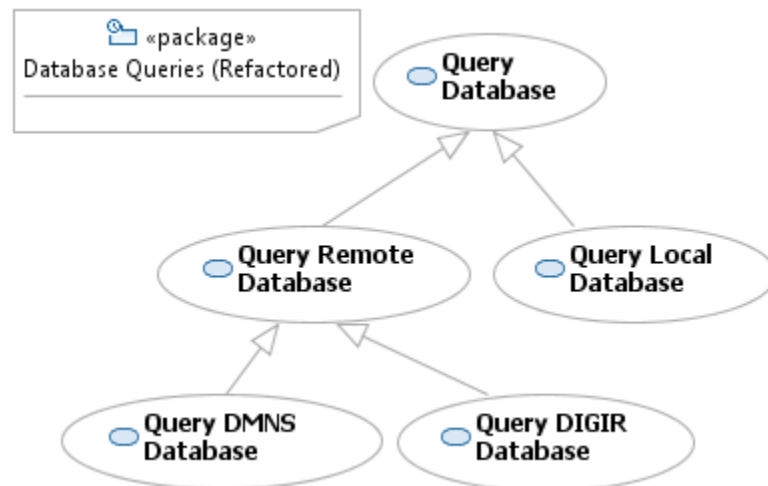


Figure 28: Use case model of Database Queries subsystem after applying the Extension to Generalization Refactoring

Database Integrator Subsystem

In the *Database Integrator* use case model, use case *Update Collections Data* includes use case *Edit Collections Data*, which is neither directly associated with any

actor nor *included* any other use case. This matches antipattern a4. The same antipattern is similarly matched by use cases *Run QC Tests* and *Upload DBG and UCOM Data*. Therefore, two instances of this antipattern are present in the *Database Integrator* use case model. Refactoring r7 is applied on the use case model. Both of the antipattern instances are refactored in a single execution of the model transformation. This results in the deletion of use cases *Edit Collections Data*, and *Upload DBG and UCOM Data*.

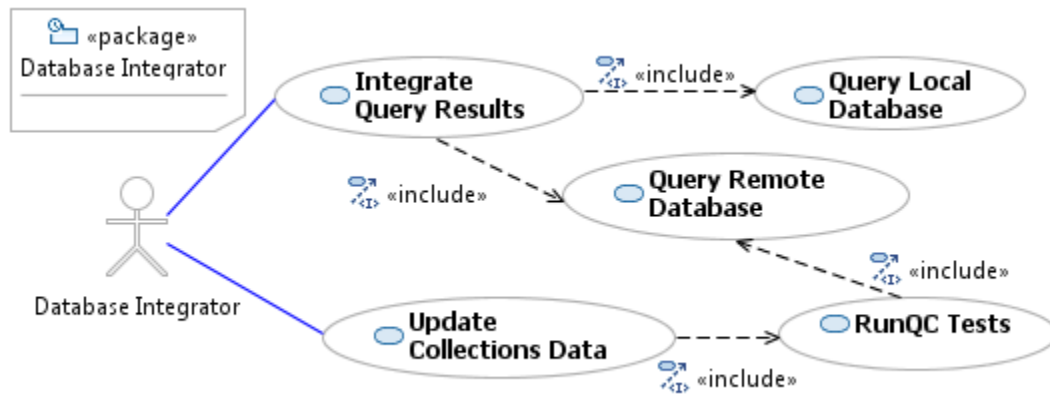


Figure 29: Use case model of Database Integrator subsystem after applying the Drop Functional Decomposition refactoring

In the refactored use case model in Figure 29, use case *Run QC Tests* is *included* by only one use case, *Update Collections Data*, and not directly associated with any actor. This is again matches antipattern a4. Therefore, the *Database Integrator* use case model must go through a second iteration of refactoring. Since use case *Run QC Tests* includes *Query Remote Database*, refactoring r8 is applied. This results in the deletion of use case *Run QC Tests* and introduction of *include* relationship between use cases *Update*

Collections Data, and *Query Remote Database*. Figure 30 presents the resulting use case model after the second iteration of refactoring.

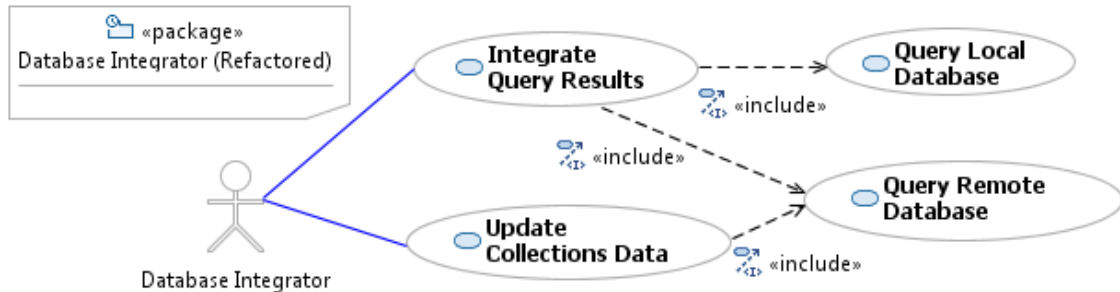


Figure 30: Use case model of Database Integrator subsystem after applying the Drop Functional Decomposition with Include refactoring on the use case model in Figure 29

Database Edits Subsystem

Use case *Edit Collections Data* was merged into use case *Update Collections Data* in the previous refactoring step. *Edit Collections Data* is also part of the *Database Edits* use case model. Therefore, use case *Edit Collections Data* must be replaced by use case *Update Collections Data* in the *Database Edits* use case model. Actors *Data Editor* and *Database Integrator* are now associated with use case *Update Collections Data*, which *extends* use case *Geocode Specimen*. This matches antipattern a2. Analysis of the *Geocode Specimen* use case description revealed that performing database updates is part of its required functionality. Therefore, the *extend* relationship is replaced by an *include* relationship directed from use case *Geocode Specimen* to use case *Update Collections Data*. The incorrect *extend* relationship is a mistake made by the modeler rather than the cause of an antipattern a2 instance. The description of actor *Data Editor* suggests that it represents the data editing role of actor *Geocoder*, which is indirectly associated with use

case *Update Collections Data* through use case *Geocode Specimen*. Therefore, actor *Data Editor* is redundant and must be deleted from the use case model.

Use case *Find Locality* is included by *Geocode Specimen* and not associated with any actor. This matches antipattern a4. Refactoring r7 is applied on the use case model; thus, resulting in use case *Find Locality* being deleted. Figure 31 presents the refactored use case model of the *Database Edits* subsystem.

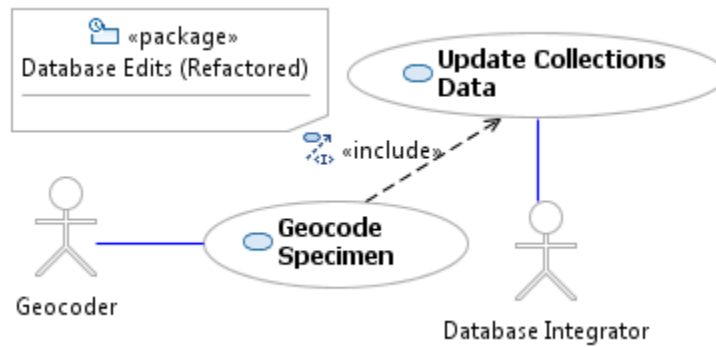


Figure 31: Use case model of Database Edits subsystem after applying the Drop Functional Decomposition refactoring

Administrative Process Subsystem

In the *Administrative Process* use case model, actors *Database Administrator* and *ArcIMC Administrator* are associated with the same set of use cases. This matches antipattern a8. Analyzing the use case descriptions of use cases *Backup Process*, *Restore Process* and *Install Software Updates* reveals that the services provided by them are too general for either of the actors, *Database Administrator* and *ArcIMC Administrator*. Actor *Database Administrator* was involved in backing up and restoring bio diversity data, and installing database updates, whereas actor *ArcIMC Administrator* was involved

in backing up and restoring application code, and installing code updates. Therefore, refactoring r15 is applied on this antipattern instance. This results in use cases *Backup Process*, *Restore Process*, and *Install Software Updates* split into two use cases, each of which provide appropriate service to actors *Database Administrator* and *ArcIMC Administrator*. The new use cases must be renamed appropriately by the modeler. Figure 32 presents the refactored use case model of the *Administrative Process* subsystem.

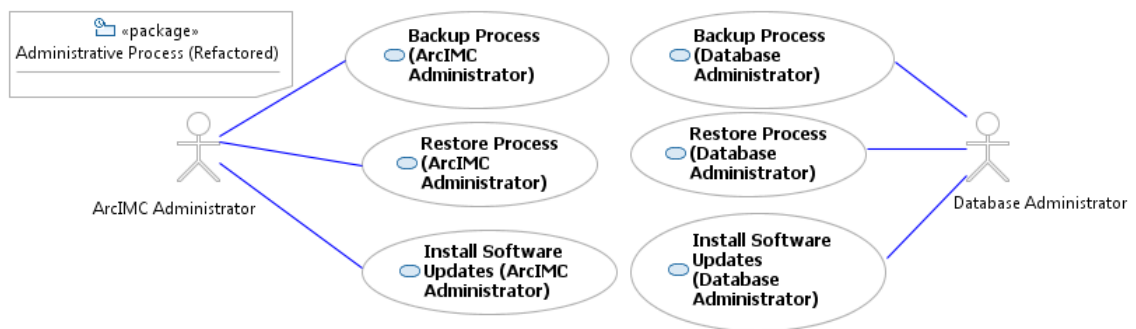


Figure 32: Use case model of Administrative Process subsystem after applying the Split UCs refactoring

Merged View

The *Database Queries* and *Database Integrator* use case models contain two overlapping use cases, *Query Remote Database* and *Query Local Database*. Therefore, their refactored use case models must be merged and considered for further refactoring opportunities. Figure 33 shows the merged use case model.

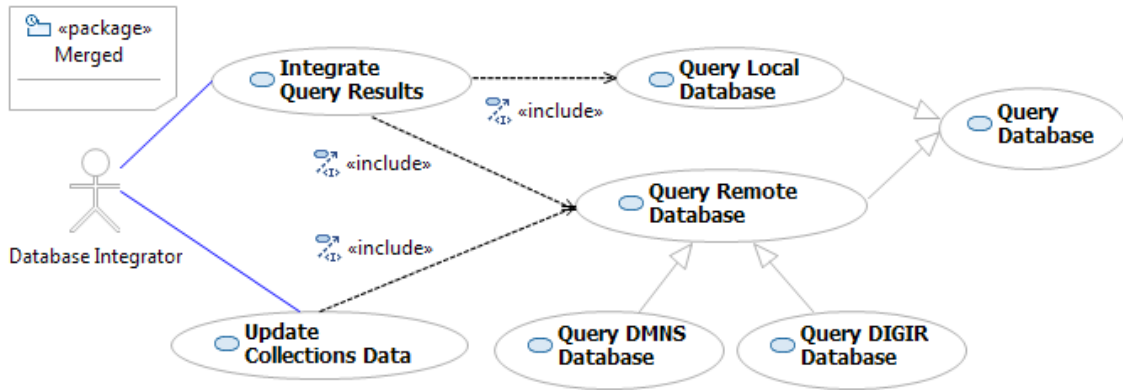


Figure 33: Merged use case model of Database Queries and Database Integrator subsystem

The refactoring performed for the *Database Queries* subsystem replaced inappropriate *extend* relationships with *generalization* relationships. Actor *Database Integrator* is indirectly associated with *generalized* use case *Query Remote Database*. This matches antipattern a1. Refactoring r1 is applied; thus, resulting in use case *Query Remote Database* set to *abstract*. The alternative refactoring for this antipattern, r2, cannot be applied because the association between actor *Database Integrator* and use case *Query Remote Database* is indirect. Figure 34 shows the refactored use case model.

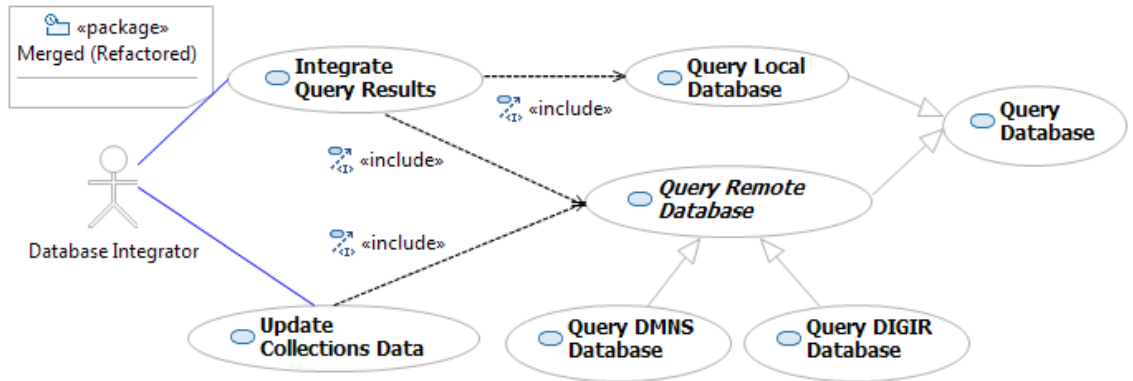


Figure 34: Use case model of Database Queries and Database Integrator subsystem after applying the Concrete to Abstract refactoring

Table 5 summarizes the antipatterns matched in the use case models of MAPSTEDI, and the refactorings applied for quality improvement. The modeler must ensure that a refactoring is behavior preserving before applying it. This can be done by consulting the use case descriptions of the use case model elements involved in the antipattern. In the case of the antipattern a2 match in the *Database Edits* use case model, the relevant use case descriptions suggested presence of mistakes made by the modeler rather than that of an antipattern instance.

Table 5: Antipatterns matched in the use case models of MAPSTEDI, and the refactorings applied

Package	Antipattern	Refactoring
Database Access	Multiple actors associated with one use case	Generalize Actors
Database Queries	Functional Decomposition: Using the <i>extend</i> relationship	Extension to Generalization
Database Integrator	Functional Decomposition: Using the <i>include</i> relationship	Drop Functional Decomposition
	Functional Decomposition: Using the <i>include</i> relationship	Drop Functional Decomposition having Inclusion
Database Edits	Accessing an <i>extension</i> use case	-
	Functional Decomposition: Using the <i>include</i> relationship	Drop Functional Decomposition
Administrative Process	Multiple actors associated with one use case	Split UCs
Database Queries and Database Integrator	Accessing a <i>generalized concrete</i> use case	Concrete to Abstract

3.3 Evaluation

The results of the case study show that model transformations detected the same set of antipatterns matched in [19]. The target models produced by the model transformations were found to be consistent with the refactored use case models of individual MAPSTEDI subsystems presented in [19]. Moreover, the problems caused by the antipatterns were resolved in the target models, thus improving their understandability and correctness. However, a discrepancy was noticed in the merged use case model of *Database Queries* and *Database Integrator* subsystems. The target model in Figure 33 shows an *include* relationship from use case *Update Collections Data* to use case *Query Remote Database* whereas, in [19] an *extend* relationship is shown between them. The discrepancy was investigated by consulting the use case descriptions, and interviewing

the original author. The *extend* relationship from use case *Update Collections Data* to use case *Query Remote Database* was determined to be a human error. This suggests that the model transformation approach is less error prone compared to the manual antipattern matching approach in [19]. MAPSTEDI is a small scale system containing 20 use cases; the usage of the antipattern matching approach on MAPSTEDI resulted in one error. For large scale systems containing thousands of use cases, the antipattern matching approach may be more error prone. Therefore, the model transformation approach is more efficient, and appropriate for large scale software systems.

CHAPTER 4

AUTOMATED TRANSFORMATION OF USE CASE

MAPS TO UML 2 ACTIVITY DIAGRAMS

This chapter proposes a model transformation approach to transform a given UCM into a UML 2 Activity Diagram (AD). The model transformation approach will systematically produce a consistent and accurate representation of UCMs in the form of ADs. Defining a formal model transformation approach has the obvious advantage of avoiding human errors which would otherwise be injected if the transformation was performed manually. In OO software development projects, the generated ADs will greatly ensure that the developed end system accurately represents the behavior modeled originally in the UCM diagrams.

The remainder of this chapter is organized as follows. Section 4.1 proposes mappings from UCM to UML 2 AD notation. Section 4.2 presents the most critical transformation rules and their implementations. Section 4.3 gives presents two case studies to illustrate the transformation approach. The first pertains to an elevator control system and, the second pertains to a mock system. Section 4.4 describes verification of the case studies.

4.1 UCM to UML 2 AD mappings

This section outlines the proposed mappings between the UCM and UML 2 AD notations. The proposed mappings shown in this section were verified by Dr. Jameleddine Hassine, a prominent researcher in the field of UCM and a professor of Software Engineering.

UCMs and ADs share similar concepts. The definitions of UCM constructs given by Buhr and Casselman [39] were used. For AD constructs, the definitions provided in the OMG UML 2.2 specification [141] were used. The definitions obtained for UCM and AD constructs were used to propose mappings between the UCM and AD notations.

A UCM is composed of one or more paths. Each path describes a particular scenario. An activity in an AD can also contain multiple flows of control. Hence, mappings between UCMs and ADs are proposed. Start points which represent the initiation of a UCM path are mapped to UML initial nodes. UCM end points which represent the termination of UCM path are mapped to UML final nodes.

The OMG UML 2.2 specification defines an *opaque action* as “an action with implementation-specific semantics”. Since UCM responsibilities are high level descriptions of system behaviour, they are mapped to opaque actions. Buhr and Casselman [39] define a timer as “a special kind of responsibility along a path that takes up real time without taking up processing resources”. Based on this analogy timers are mapped to opaque actions as well, similar to the mapping of responsibilities except that a

‘No Action’ label is appended to the timer’s notation to distinguish it from other opaque actions. UCM failure points are defined as “points where a path may end abnormally, due to some failure in the underlying system” [39]. They simply indicate the possible occurrence of a failure or exception; thus, they are mapped to opaque actions. A label ‘Handle Exception’ is appended to the failure point’s name in order to distinguish it from other opaque actions.

UCM concurrency and branching constructs, AND-fork and AND-join, are intuitively mapped to their AD counterparts, fork node and join node, respectively. It should be noted that concurrent control flows in ADs are required to synchronize at a join node; however, UCMs have no such restriction [10]. UCM branching constructs, OR-fork and OR-join are intuitively mapped to their AD counterparts, decision node and merge node, respectively.

The UCM elements which are bound to components (teams, objects, processes, actors, and agents) are grouped into activity partitions. This mapping decision is made since their purpose is to group related activity nodes together and to represent organizational units such as classes [159]. The difference between these notations is that UCM components cannot share elements (responsibilities, timers, failure points, etc.) whereas ADs have no such restriction. ADs allow activity partitions to overlap, enabling them to share nodes and edges. Hierarchical decomposition of activity partitions in ADs is similar to that of components in UCMs. In order to determine which type of component (actor, process, object, etc.) they correspond to, we suggest their names be appended with

the type of component they correspond to. Names of activity partitions that correspond to generic components (of no specific type) are appended with an '(Other)' label.

Stubs which represent nested UCMs are mapped to structured activities, which cannot share nodes and edges with other structured activities. This mapping decision is made because stubs are individual UCMs, by themselves, which do not share elements (responsibilities, timers, failure points, etc.) with parent or child maps. It should be noted that components inside a stub will be ignored by our mapping, since structured activities cannot include activity partitions. However, nesting of structured activities is allowed as is the case with stubs in UCMs. In order to prevent loss of information while using this mapping, UCM designers should model stubs such that they are contained within a component.

UCM waiting points are points along a path that indicate that execution flow must wait for events along another path [39]. There is no such notation in ADs that can allow a control flow to wait for another one. We propose to use merge nodes with labels appended by 'Wait', to depict such behavior in a flow. It should be noted that the end point that is connected to a waiting point is discarded during the mapping. Otherwise, it would be mapped to an activity final node, which would be connected to a merge node (waiting). This mapping decision is made since a final node stops a flow in an activity. A visual summary of the mappings is shown in Figure 35.

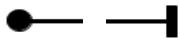



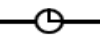
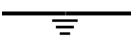
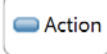


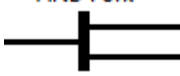

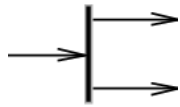
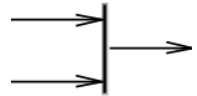
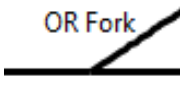
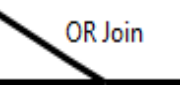
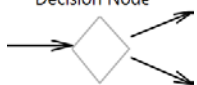
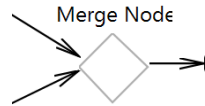






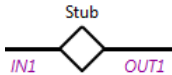


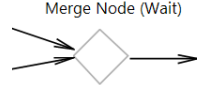
UCM	AD
<p>Start Point End Point</p> 	<p>Initial Node Final Node</p>  
<p>Responsibility Timer</p>   <p>Failure Point</p> 	  
<p>AND Fork</p>  <p>AND Join</p> 	<p>Fork Node</p>  <p>Join Node</p> 
<p>OR Fork</p>  <p>OR Join</p> 	<p>Decision Node</p>  <p>Merge Node</p> 
<p>Actor Team</p>   <p>Process Agent</p>   <p>Object</p> 	<p>Activity Partition</p> 
<p>Stub</p> 	<p>«structured» Structured Activity</p> 
<p>Waiting Place</p> 	<p>Merge Node (Wait)</p> 

Figure 35: Mapping of UCM to UML 2 AD notation

4.2 Transformation Rules

The proposed mapping was implemented using ATL [86][175], a model transformation language. ATL provides a hybrid of declarative and imperative programming styles for defining mappings between source and target models. As such, both programming capabilities were used to implement this transformation. The rules were written against the UCM metamodel and UML 2 metamodel shown in Appendix A and Appendix B, respectively. The remainder of this section presents the different types of rules that were developed.

4.2.1 Entry point and Matched Rule

The transformation begins with executing an *entrypoint* rule *Main*. The entrypoint rule (see Listing 21) transforms UCM nodes, edges, components and stubs to their corresponding AD notation by invoking called rules (see Listing 24). Once the entrypoint rule finishes execution, the matched rule, *URNDefinition_To_UMLPackage*, is implicitly invoked. Matched rules define the transformation process in a declarative manner. *URNDefinition_To_UMLPackage* was created (see Listing 22) to map the root node of a source UCM to the corresponding one of a target AD.

```

entrypoint rule Main() {
  do {
    for(ucmNode in thisModule.ucmNodes){
      thisModule.TransformNode(ucmNode);
    }
    for(ucmEdge in thisModule.ucmEdges){
      thisModule.TransformEdge(ucmEdge);
    }
    for(component in thisModule.ucmComponents){
      thisModule.TransformComponent(component);
    }
    for(stub in thisModule.rootMapStubs) {
      thisModule.TransformStub(stub);
    }
  }
}

```

Listing 21: The entry point rule

```

rule URNDefinition_To_UMLPackage {
  from d: UCM!"urn:URNSpec"
  to p: UML!Package (
    packagedElement <- a
  ),
  a: UML!Activity (
    name <- thisModule.rootUCM.name,
    node <- thisModule.umlNodes,
    edge <- thisModule.umlEdges,
    group <- thisModule.umlGroups
  )
  do {
    p.debug('Transformation done!');
  }
}

```

Listing 22: The matched rule

4.2.2 Lazy Rules

In ATL, rules that do not state parameters are given the modifier *lazy*. Lazy rules facilitate the transformation process in the same manner as matched rules. Unlike matched rules, lazy rules only execute when called by other rules. Although they are defined without parameters, they require parameters to be passed while invoking them. The *from* and *to* blocks in a lazy rule are declarative statements that specify the source and target instance respectively. The following are the lazy rules that have been

implemented in order to transform various UCM notations into UML 2 AD notations.

Listing 23 outlines the lazy rules and briefly explains the purpose of each one.

Transforming <i>start point</i> into <i>end node</i> .	Transforming <i>end point</i> to <i>final node</i> .
<pre>lazy rule StartPoint_To_InitialNode { from p: UCM!"ucm::map::StartPoint" to n: UML!InitialNode (name <- p.name) }</pre>	<pre>lazy rule EndPoint_To_FinalNode { from p: UCM!"ucm::map::EndPoint" to n: UML!ActivityFinalNode (name <- p.name) }</pre>
Transforming <i>responsibility</i> to <i>action</i> .	Transforming <i>timer</i> to <i>action</i> .
<pre>lazy rule Responsibility_To_OpaqueAction { from r: UCM!"ucm::map::RespRef" to a: UML!OpaqueAction (name <- r.respDef.name) }</pre>	<pre>lazy rule Timer_To_OpaqueAction { from t: UCM!"ucm::map::Timer" to n: UML!OpaqueAction (name <- t.name + ' (No Action)') }</pre>
Transforming <i>waiting place</i> node to <i>merge</i> node.	Transforming <i>failure point</i> to <i>action</i> .
<pre>lazy rule WaitingPlace_To_MergeNode { from w: UCM!"ucm::map::WaitingPlace" to n: UML!MergeNode (name <- w.name + ' (Wait)') }</pre>	<pre>lazy rule FailurePoint_To_OpaqueAction { from f: UCM!"ucm::map::FailurePoint" to n: UML!OpaqueAction (name <- f.name + ' (Handle Failure)') }</pre>
Transforming <i>AND-fork</i> to <i>fork</i> node.	Transforming <i>AND-join</i> to <i>join</i> node.
<pre>lazy rule AndFork_To_ForkNode { from f: UCM!"ucm::map::AndFork" to n: UML!ForkNode (name <- f.name + ' (Fork)') }</pre>	<pre>lazy rule AndJoin_To_JoinNode { from f: UCM!"ucm::map::AndJoin" to n: UML!ForkNode (name <- f.name + ' (Join)') }</pre>
Transforming <i>OR-fork</i> to <i>decision</i> node.	Transforming <i>OR-Join</i> to <i>merge</i> node.
<pre>lazy rule ORFork_To_DecisionNode { from o: UCM!"ucm::map::OrFork" to n: UML!MergeNode (name <- o.name + ' (Decision)') }</pre>	<pre>lazy rule ORJoin_To_MergeNode { from o: UCM!"ucm::map::OrJoin" to n: UML!MergeNode (name <- o.name + ' (Merge)') }</pre>

Listing 23: Lazy rules

It should be noted that in rule *AndJoin_To_JoinNode* the target object's type is fork node rather than join node. This is because the Eclipse UML 2 tools do not contain notation for join node. The tool also lacks notation for decision nodes. Hence, in rule *ORFork_To_DecisionNode* the *to* block defines an instance of merge node rather than decision node.

4.2.3 Called Rules

Called rules describe part of the transformation process in an imperative manner. They are referred as called since they must be explicitly invoked by the developer. Called rules may contain a *using* block where local variables may be defined. A *do* block can be used to write imperative statements. The last statement of the *do* block must return the target model instance. In the *using* and *do* blocks of these rules, helper functions can be invoked. Due to space limitations only three called rules are presented in Listing 24.

<p>Initializes an <i>activity partition</i> given a <i>component</i></p> <pre> rule InitUmlGroup(compRef: UCM!"ucm::map::ComponentRef") { using { groupName: String = compRef.groupName(); groupNodes: Sequence(UML!Node) = compRef.getUmlNodes(); } to a: UML!ActivityPartition (name <- groupName, node <- groupNodes) do { if(compRef.hasNoChildren()) { a; } else { a.subpartition <- compRef.getUmlSubGroups(); a; } } } </pre>
<p>Initializes a <i>structured activity node</i> given a <i>stub</i></p> <pre> rule InitStaticStrAct(stub:UCM!"ucm::map::Stub") { using { map: UCM!"ucm::map::UCMmap" = stub.getMap(); source: UCM!"ucm::map::NodeConnection" = stub.firstPredecessor(); target: UCM!"ucm::map::NodeConnection" = stub.firstSuccessor(); } to a: UML!StructuredActivityNode (incoming <- source.getUmlEdge(), outgoing <- target.getUmlEdge(), node <- map.getNodes(), edge <- map.getEdges(), name <- stub.name) do { thisModule.ProcessStrActElements(stub, a); if(map.hasNoStubs()) { a; } else { thisModule.ProcessNestedMaps(map, a); a; } } } </pre>
<p>Initializes a <i>control flow</i> given a <i>node connection</i></p> <pre> rule InitUmlEdge(ucmEdge: UCM!"ucm::map::NodeConnection") { using { umlSource: UML!Node = ucmEdge.getSourceUmlNode(); umlTarget: UML!Node = ucmEdge.getTargetUmlNode(); label: String = ucmEdge.getLabel(); } to e: UML!ControlFlow (source <- umlSource, target <- umlTarget, name <- label) do { thisModule.AddEdgeMap(ucmEdge, e); e; } } </pre>

Listing 24: Called rules

4.2.4 Helpers

A number of ATL helpers were written to facilitate the transformation process. The helper *hasNoParent* in Listing 25 determines whether a UCM component (actor, process, agent, etc.) is nested in another component or not. The helper *isDiscardable* in Listing 25 determines whether a UCM node may be discarded during the transformation process.

```
helper context UCM!"urncore::Component" def: hasNoParent(): Boolean =
  self.contRefs>first().parent.oclIsUndefined();
helper context UCM!"ucm::map::PathNode" def: isDiscardable() : Boolean =
  self.oclIsTypeOf(UCM!"ucm::map::DirectionArrow") or
  self.oclIsTypeOf(UCM!"ucm::map::EmptyPoint") or
  self.oclIsTypeOf(UCM!"ucm::map::Stub");
```

Listing 25: Helper rules

The transformation algorithm was implemented using a total of 45 transformation rules. Due to space restrictions, this chapter only presents the most critical 17 rules. The entire ATL source code is available to the interested reader for download at [94].

4.3 Case Studies

In this section two case studies are presented to illustrate the proposed transformation approach.

4.3.1 Elevator Control System

The implemented ATL transformation is applied to the UCM (Figure 36) of an Elevator Control System (ECS), which is available at [8]. The UCM was adapted from

“Designing Concurrent, Distributed and Real-Time Applications with UML” [74]. It represents the functionality of an ECS that controls one or more elevators. The two main responsibilities of the system are to respond to elevator calls from users, and to manage the motion of the elevators between floors.

A use case begins with a request from the user to call the elevator to go to above or below levels. The request gets queued with other call requests. Depending on the state of the elevator whether it is stationary or moving, the system will control motor actions to move the elevator appropriately. Once the elevator approaches a requested floor, the motor stops, the door opens, and the corresponding call request is removed from the queue.

This model was selected since it includes most of the UCM notational set and represents a complex scenario with multiple alternates. The source model (Figure 36) was provided as input to the ATL transformation algorithm defined, which resulted in the generation of the AD shown in Figure 37. The ATL source code, source and target models are available to the interested reader for download at [94].

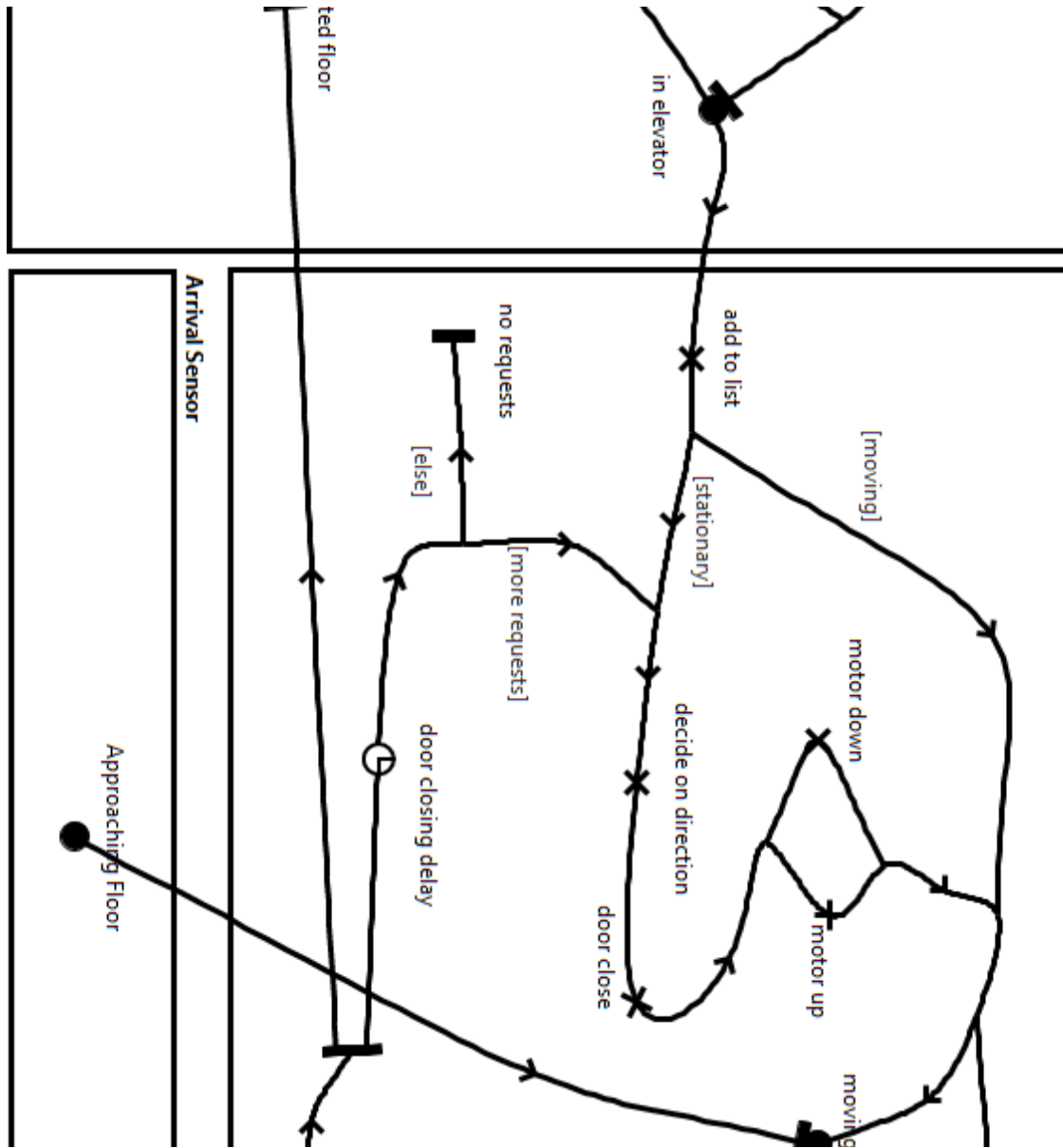


Figure 36: Elevator Control System source UCM

notation such as objects, processes, joins and failure points were not part of the previous example. This example involves all elements of the UCM and AD notation given in Figure 35. The remainder of this section describes the source and target model involved in this example.

Source Model

The source model (Figure 38) is applied as input to the ATL transformation. The UCM starts at start point *SP1*, performs several responsibilities and ends at *EPI*. The responsibilities can be bounded to a particular component or remain unbounded. Responsibility *RU* is an unbounded responsibility, whereas the remaining responsibilities are bounded to their respective components. Responsibilities *RO* and *RT* are bound to components *CO* and *CT*, respectively. Responsibility *RAg* and failure point *FP* are bound to the agent component *CAg*. The actor component *CAC* contains a nested process component *CP*. Start point *SP2*, responsibility *RAC*, end point *EP2*, and waiting point *WP* are bound to *CAC*, and responsibility *RP* is bound to *CP*. The path first performs *RU* after which it forks into two concurrent paths at AND-fork *AF*. They concurrently perform responsibilities *RO* and *RT* in components *CO* and *CT*, respectively. They synchronize at AND-join *AJ*, after which the path enters stub *NM* (Figure 39). It performs responsibility *RS* and waits for 5 seconds (timer), and reenters the main UCM. The remainder of the path branches into alternate paths at OR-Fork *OF* based on guard conditions *C1* and *C2*. If *C1* is satisfied, the path waits at waiting point *WP* for actor *CAC* to perform *RAC*. Once the wait is over, process *CP* executes responsibility *RP*. If *C2* is satisfied, *RAg* is performed by agent *CAg*. The failure point *FP* bound to *CAg* indicates an erroneous situation whose

occurrence can terminate the path. The alternate paths merge at OR-join *OJ*, after which the path terminates at *EP1*.

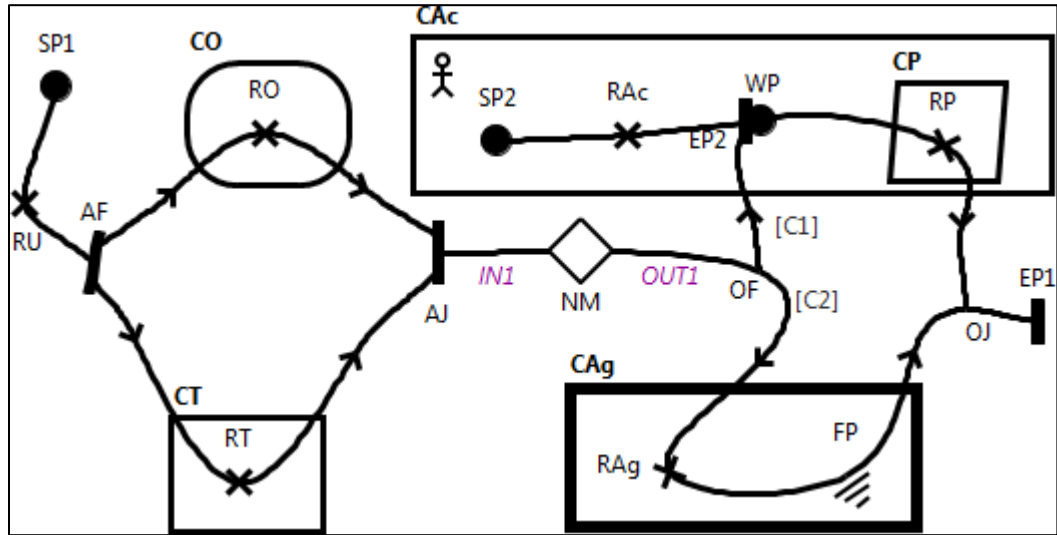


Figure 38: Mock System source UCM

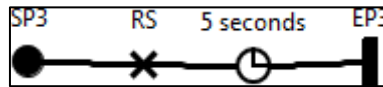


Figure 39: Mock System Stub NC

Target Model

The target model in Figure 40 is the output of the ATL transformation when Figure 38 is given as input. It can be seen that components from the source model have been transformed to activity partitions. Their names have been appended with the type of component it corresponds to. For example, partition *CO* is appended with ‘Object’ in parentheses. Partition *CAc* contains a sub partition *CP*, as consistent with corresponding component in the source model. Responsibilities from the source model were transformed

to actions in the target model. It should be noted that bounded responsibilities from the source model have been grouped into corresponding partitions. Action *RU*, decision node *OF*, merge node *OJ*, structured activity *NM*, fork node *AF*, and join node *AJ* are not grouped into any partition, as they were unbounded in the source model. Structured activity *NM* includes *SP3*, *RS*, a timer and *EP3*, consistent with the elements of the stub *NM* in the source model. The arrows along the map are discarded during the transformation since they are already depicted by AD control flow.

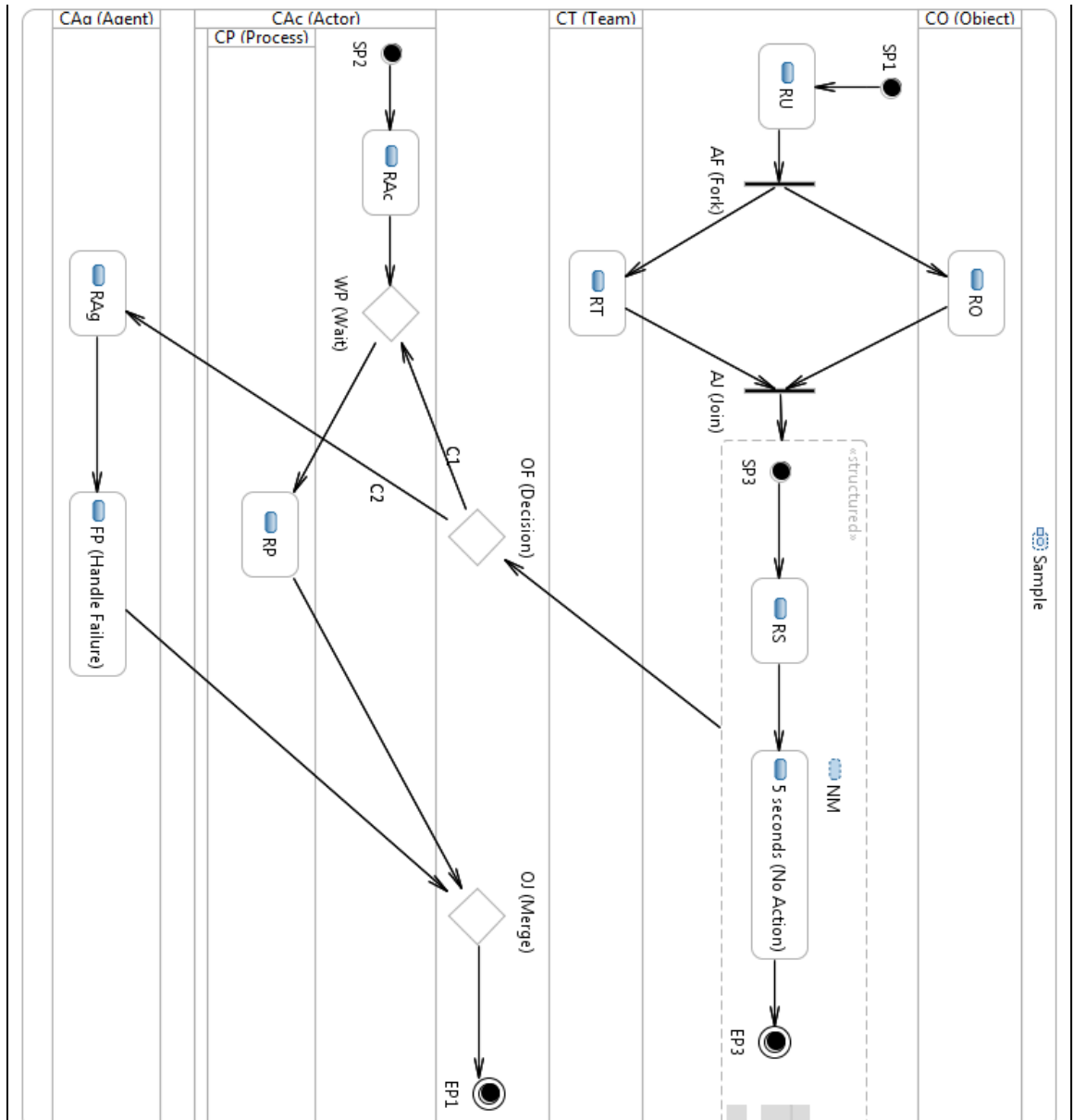


Figure 40: Mock System target AD

4.4 Target Model Verification

The target model was thoroughly inspected and verified by three Software Engineering professors at the host institution. The proposed mapping (Figure 35) was

given to them along with the source and target models. Reviewers indicated confusion while distinguishing between decision nodes and merge nodes. This confusion is due to the fact that the Eclipse UML 2 tools do not include separate notation for decision nodes. Merge nodes are intended to be used in place of decision nodes. Hence, to avoid this confusion labels are placed on their respective notations. Another reviewer indicated confusion while interpreting edges coming in and out of fork and join nodes. This was found to be a layout issue. The transformation results in the model elements being placed in a default layout. The target model was manually realigned to clear the confusion. The same reviewer indicated that the proposed mapping did not consider dynamic stubs. Hence, a mapping for dynamic stubs was implemented in ATL. This can be found in the available source code.

CHAPTER 5

DERIVING UML 2 SEQUENCE DIAGRAMS FROM USE

CASE MAP SCENARIO SPECIFICATIONS

This chapter presents a traceable mapping for transforming use case scenarios from UCM to UML 2 SD notation. Traceability helps in assessing the validity and completeness of requirements [147]. Similar to SDs, UCMs can be modeled at varying levels of abstractions. High level UCMs depict components at abstract levels of granularity; such UCMs can easily be verified by the clients or end users, who are usually not concerned about the composition of system components. A high level UCM can also be depicted sans components, which again promotes the verification of scenarios by clients. Detailed UCMs depict scenario interactions bound to a particular system component. This promotes architectural reasoning at the functional requirements phase of a software development process, and serves a reference point for the architecture phase. UCMs describe scenarios at a higher level of abstraction compared to SDs. UCMs do not illustrate how system components interact with each other, whereas SDs do so via message interactions. The approach proposed in this chapter will ease the refinement of scenarios from UCM to UML 2 SD notation. The chapter also defines model transformation rules that can semi-automate the transition, and serve as a start-up point for deriving SDs from UCMs.

The remainder of this chapter is organized as follows. Section 5.1 presents mappings from UCM to UML 2 SD notation and illustrates them with examples. In Section 5.2 we present a model transformation approach to refine UCMs to SDs. In Section 5.3, the proposed mappings are applied on a case study that pertains to an elevator control system.

5.1 UCM to UML 2 SD mappings

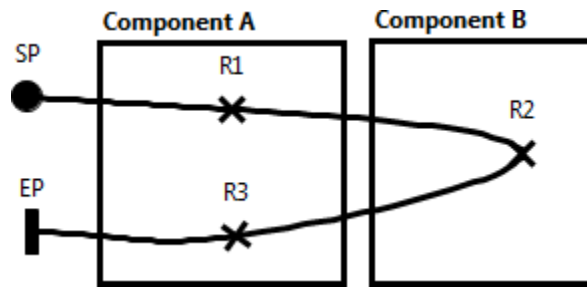
In this section, mappings between UCM and SD notation are defined and illustrated for each UCM notational element.

5.1.1 Components and Responsibilities

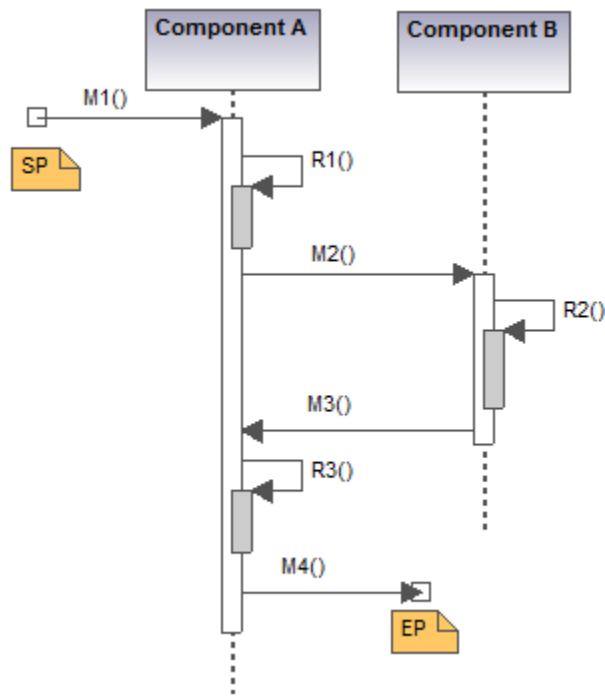
UCMs contain causal paths of responsibilities, which are superimposed over one or more components. SD lifelines can be used to represent a UCM component in a SD. The responsibilities, which are bound to components, are translated as self messages in the lifeline that corresponds to its component. The transition of a UCM path from one component to another is shown in SDs as a generic message passed from a source lifeline to a target lifeline. UCMs do not describe how components interact with each other; but SDs can do so. We leave it to the designer to decide how the components interact with each other. The message can be one of the different types (synchronous, asynchronous, creation, destruction, synchronous reply and asynchronous reply) of messages in the UML 2 SD notation. The designer may also specify any parameters that need be passed along with the message.

The unbounded start points and unbounded end points of a UCM are represented as gates, which allow external messages into or out of the SD. The flow of control from the start gates is shown as generic messages; which are external events that invoke the execution of the SD. The flow of control to the end gate is also show as generic messages; which are events that terminate the SD and are passed outside the SD (possibly to another one). Unbounded responsibilities, which are events or actions external to any of the system's components, will be ignored by this transformation. Unbounded UCMs are usually used to show high level system behavior to clients and end users. They cannot be used to derive SD; hence only detailed UCMs, which contain bounded responsibilities, can be translated to SDs.

Figure 41 illustrates the transformation of a bounded UCM. The source UCM (Figure 41(a)) contains two components, *Component A* and *Component B*, which are represented as distinct lifelines in the target SD (Figure 41(b)). Their respective responsibilities are represented as self messages in order of path traversal. Responsibilities *R1* and *R3* are translated to self messages *R1()* and *R3()*, respectively, in lifeline *Component A*, whereas responsibility *R2* is translated to self message *R2()* in lifeline *Component B*. The path transitions between *Component A* and *Component B* them are indicated by the messages *M2()* and *M3()*. The types and names of these messages can be changed by the designer as desired. *M1()* represents a message received by lifeline *Component A* from an external entity, whereas *M4()* represents a message passed to an external entity. Start point *SP* and end point *EP* are translated to start gate *SP* and end gate *EP*, respectively.



(a) Source UCM



(b) Target SD

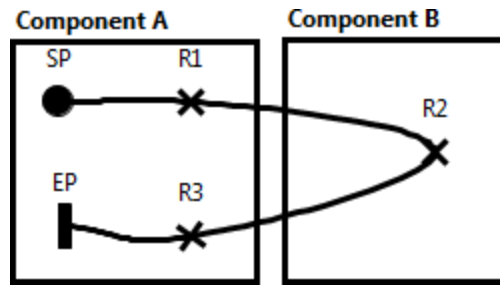
Figure 41: Mapping of components and responsibilities

Bounded Start or End Points

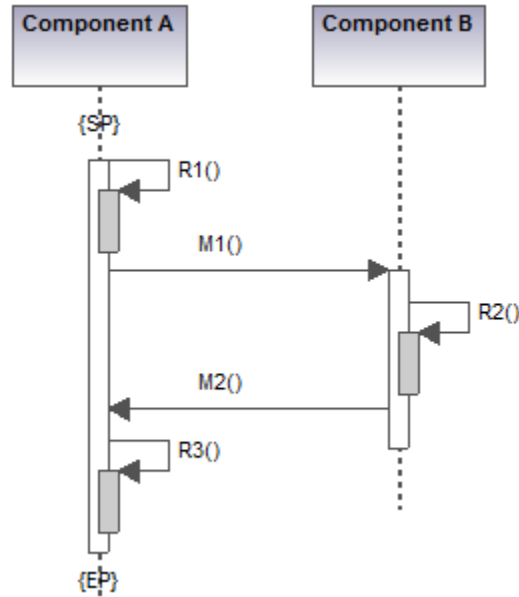
A UCM may contain start points or end points bounded to particular components. These points are not translated to gates during SD mapping. A bounded start point indicates a state, of its inclosing component, which initiates the execution of the UCM

path; hence, external messages emerging from start gates are not required. They are represented using state invariants on the enclosing component's corresponding lifeline. If a path terminates in a component, an end gate is not required in its corresponding SD. This indicates the state of the component, as a result of the termination of the path. Hence, there is no need to pass a message out of the SD (through an end gate). They are also represented as state invariants on the lifeline that corresponds to its enclosing component.

Figure 42 illustrates the mapping of a UCM containing bounded start and end points. Start point *SP* and end point *EP*, which are bounded to *Component A* in the source UCM (Figure 42(a)), are mapped to state invariants *SP* and *EP*, respectively, on lifeline *Component A* in the target SD ((Figure 42(b)).



(a) Source UCM



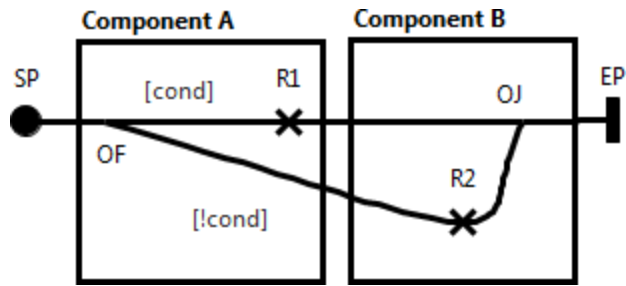
(b) Target SD

Figure 42: Mapping of bounded start and end points

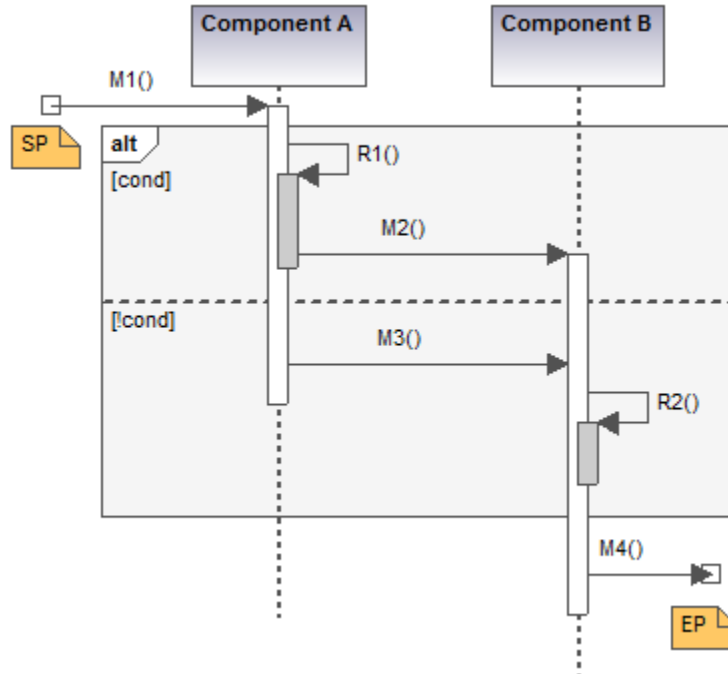
5.1.2 OR-forks

Alternate paths in a UCM emerge from OR-forks, which contain a guard condition. They are represented in a SD using the alt fragment. The fragment must contain two operators; one for the path that executes when the OR-fork guard evaluates to true, and another one for the path that executes when the OR-fork guard evaluates to false. The merging of the alternate paths at an OR-join is the termination of the alt fragment.

Figure 43 illustrates the mapping of a UCM that contains alternate paths. The source UCM (Figure 43(a)) contains an OR-fork *OF* from which alternate paths emerge. Each path has a guard condition, which must evaluate to true in order for the path to proceed. If guard *[cond]* is satisfied, responsibility *RI* is performed; otherwise control is immediately transferred to *Component B*. The alternate paths merge at OR-join *OJ*. In the target SD (Figure 43(b)), the paths emerging from *OF* are represented in an *alt* fragment. The fragment includes two operators; first one having guard *[cond]*, and second one having guard *[!cond]*. The *alt* fragment terminates either when message *M2()* is received, or when self message *R2()* is called by lifeline *Component B*.



(a) Source UCM



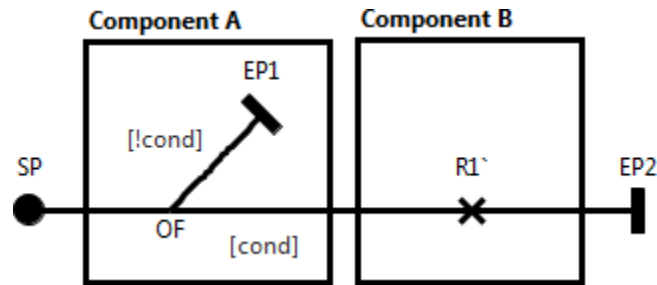
(b) Target SD

Figure 43: Mapping of alternate paths

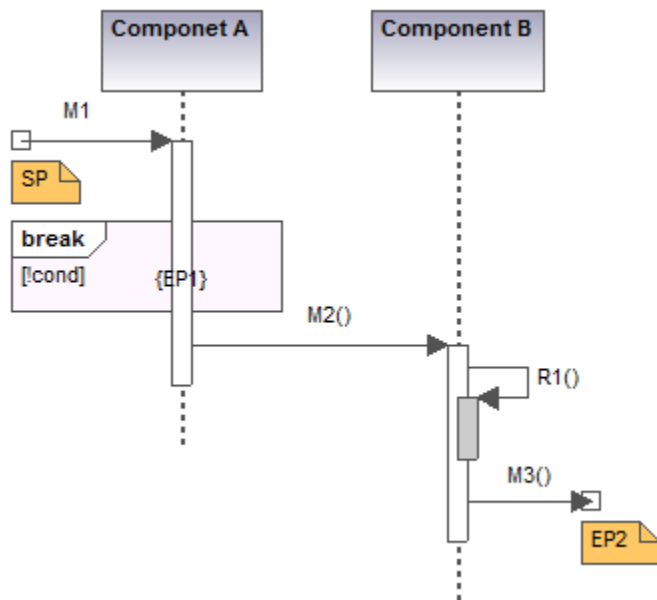
5.1.2.1 Terminating Alternate Path

An alternate path emerging from an OR-fork may immediately terminate the execution of the UCM. This is shown in a SD using the break fragment instead of an alt fragment.

Figure 44 illustrates the mapping of a UCM which contains a terminating alternate path. In the source UCM (Figure 44(a)), a terminating path emerges from OR-fork *OF* when guard *[!cond]* is satisfied. Since end point *EP1* is bound to *Component A* it represents the terminating state of *Component A*. This terminating path is enclosed by a break fragment in the target SD (Figure 44(b)). The fragment also indicates the terminating state through state invariant *EP1*.



(a) Source UCM



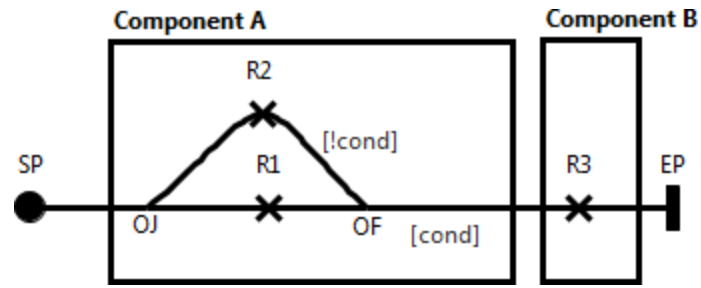
(b) Target SD

Figure 44: Mapping of terminating alternate path

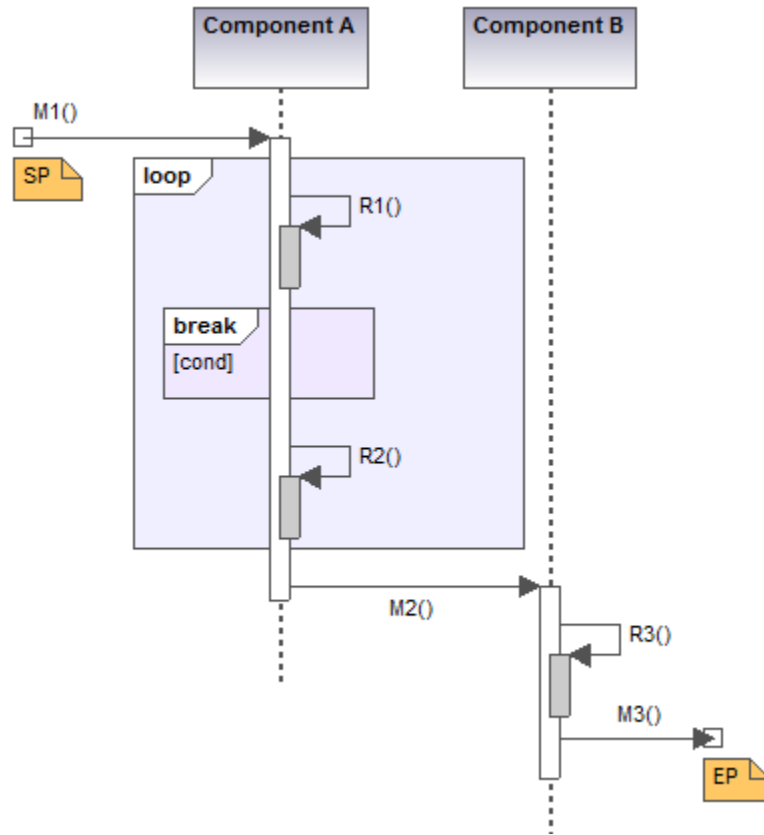
5.1.2.2 Loops

A UCM can also show repeated behavior (loops) using a combination of an OR-fork and OR-join. An alternate path emerging from an OR-fork is connected backwards to its main path using an OR-join. This is represented in SDs using a loop fragment and a *break* fragment.

Figure 45 illustrates a mapping from a UCM which contains a loop. An alternate path emerges from OR-fork *OF* in the source UCM (Figure 45(a)) when guard *[!cond]* is satisfied. This alternate path connects back to the main path at OR-join *OJ* to form a loop. In the target SD (Figure 45(b)), the UCM loop is represented using a loop fragment which encloses a *break* fragment. The *loop* fragment represents an infinite loop; it has no guard condition. The *break* fragment contains the OR-fork's guard *[cond]* which moves the path further. This allows the flow to break out of the *loop* fragment. The alternate path having guard *[cond==false]*, which loops back to main path, is represented in the remainder of the loop fragment.



(a) Source UCM



(b) Target SD

Figure 45: Mapping of a UCM loop

5.1.2.3 Loops (Alternate)

An alternate approach to mapping UCM loops uses the loop fragment in conjunction with the guard condition of the OR-fork. The break fragment is not required in this approach.

Figure 46 shows the alternate mapping of the UCM in Figure 45. The loop fragment in the target SD includes the guard *[!cond]*. Note that message *R1()* is shown twice in the target SD, first time before the loop fragment, and second time inside the loop fragment. This approach may clutter the resulting SD in case of several responsibilities preceding the OR-fork and succeeding the OR-join.

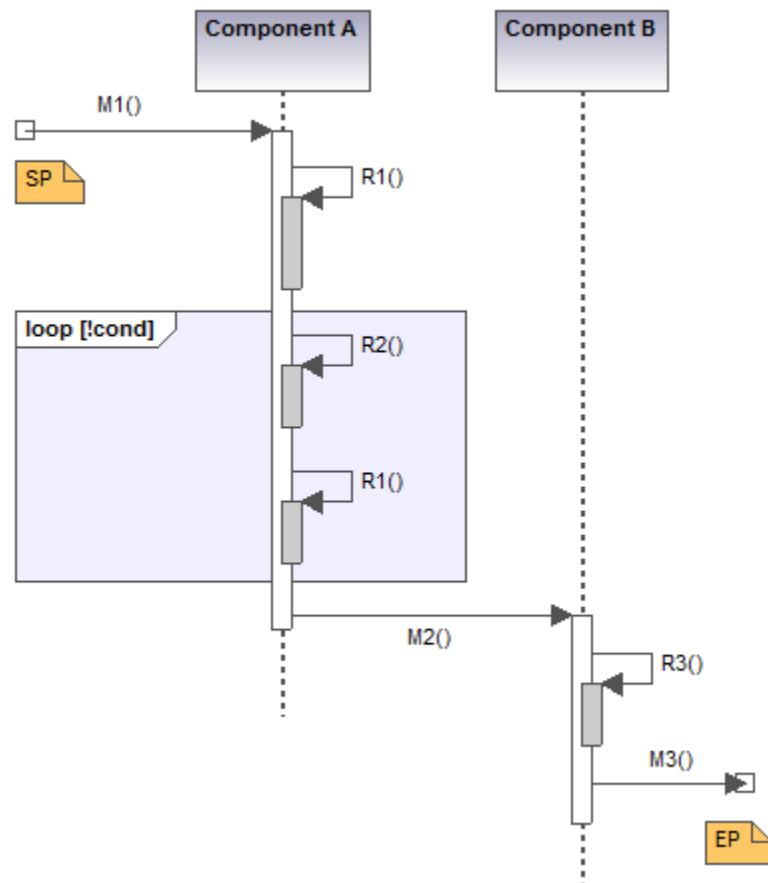
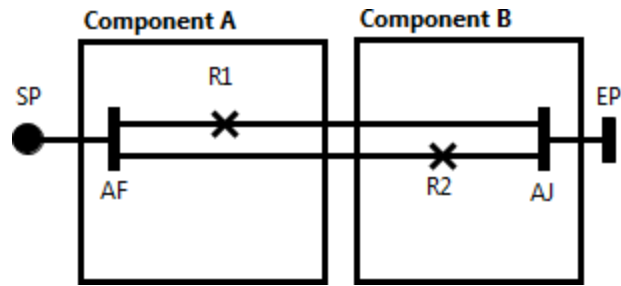


Figure 46: Alternate mapping of the UCM loop shown in Figure 45

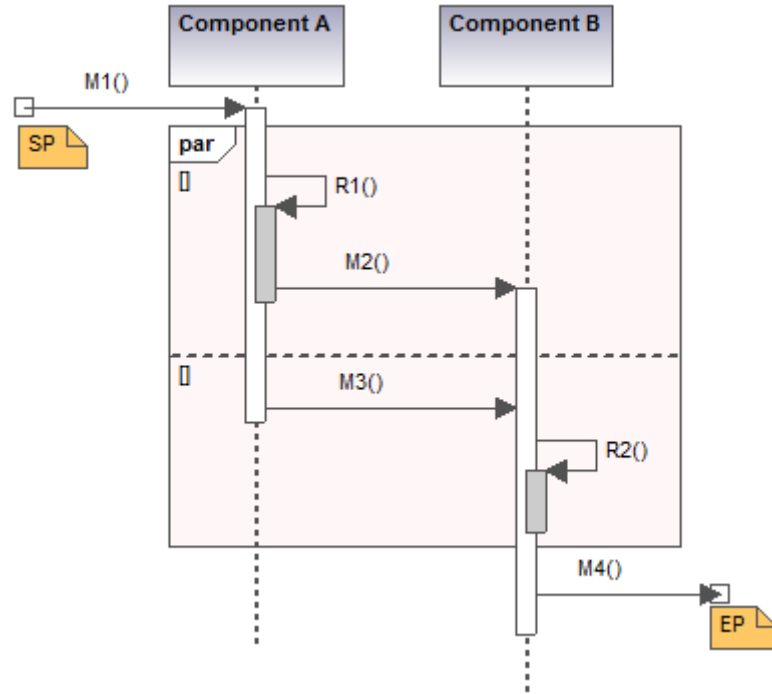
5.1.3 AND-forks

Concurrent paths in a UCM emerge from AND-forks; they can be represented in a SD using the par fragment. The fragment must contain separate operators for each path. The synchronization of concurrent paths at an AND-join translates to the termination of the par fragment.

Figure 47 illustrates the mapping of a UCM containing concurrent paths. The source UCM (Figure 47(a)) contains ANF-fork *AF* from which two parallel paths emerge. The first path performs responsibility *R1* in *Component A*, while the second path performs responsibility *R2* in *Component B*. The paths synchronize at AND-join *AJ*. The parallel paths are represented using the par fragment in the target SD (Figure 47(b)). The fragment has two operators, one for each concurrent path. The receipt of message *M2()*, and completion of message *R2()*, by lifeline *Component B* terminates the par fragment.



(a) Source UCM



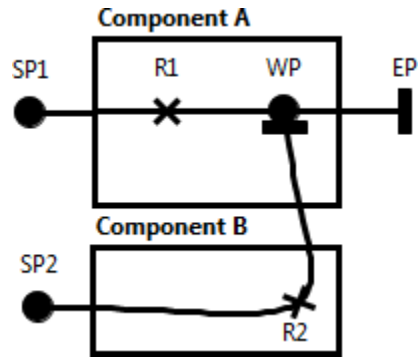
(b) Target SD

Figure 47: Mapping of concurrent paths

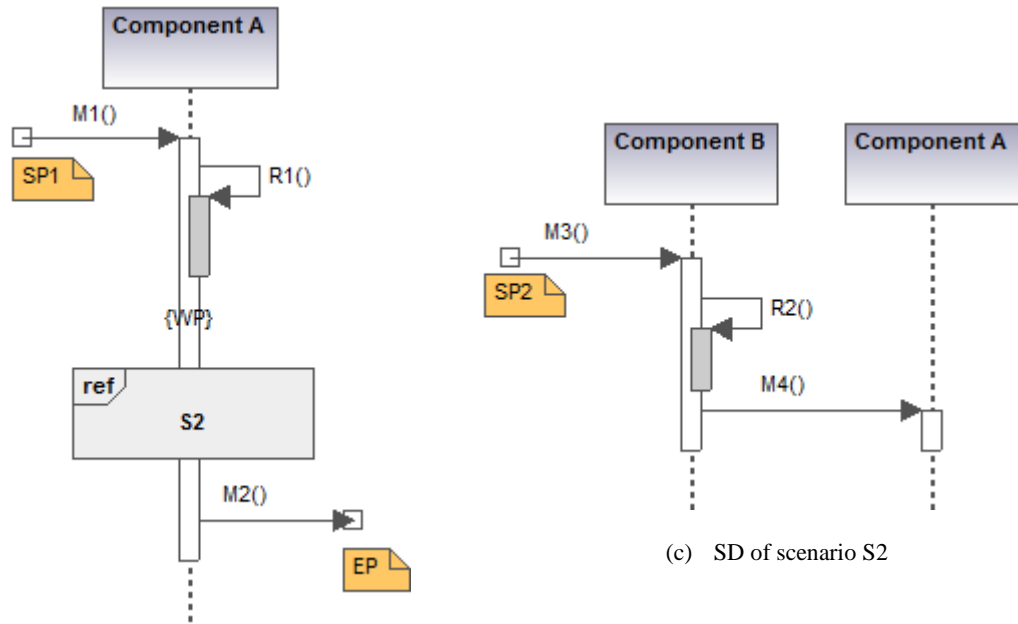
5.1.4 Waiting Point

On a waiting point, a UCM path waits for another path to finish its execution. This waiting point is represented in a SD using a state invariant. The path which is being waited for must be modeled as a separate SD, and referred from the target SD using an InteractionUse.

Figure 48 illustrates the mapping of a UCM containing a waiting point. In the source UCM (Figure 48(a)), the path emerging from *SP1* waits at waiting point *WP* for the path starting at *SP2* to finish. Let the path starting at *SP1* be scenario *S1*, and the one starting at *SP2* be scenario *S2*. *S2* is translated into a separate SD (Figure 48(c)), which is referred from the SD of *S1* (Figure 48(b)) through the InteractionUse *S2*. The state invariant *WP* on lifeline *Component A* indicates that its flow must pause until InteractionUse *S2* completes execution.



(a) Source UCM



(b) SD of scenario S1

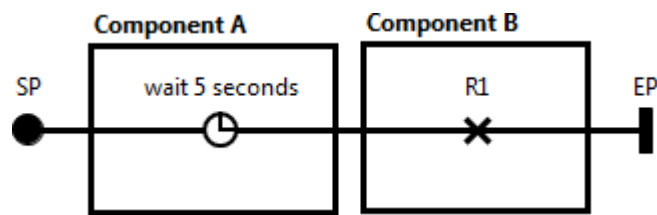
(c) SD of scenario S2

Figure 48: Mapping of waiting points

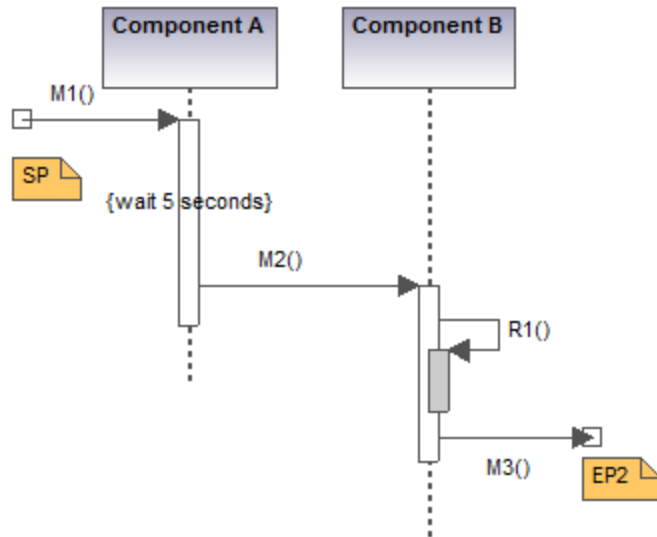
5.1.5 Timer

A UCM path waits for a specific amount of time at a timer before continuing its execution. This can be represented in a SD as a state invariant on the lifeline that corresponds to the timer's enclosing component.

Figure 49 illustrates the mapping of a UCM containing a timer. In the source UCM (Figure 49(a)), the path emerging from *SP* waits 5 seconds in *Component A* before proceeding to *Component B*. The *wait 5 seconds* timer is translated to a state invariant in the target SD (Figure 49(b)). This indicates that lifeline *Component A* must wait 5 seconds before transferring control to lifeline *Component B*.



(a) Source UCM



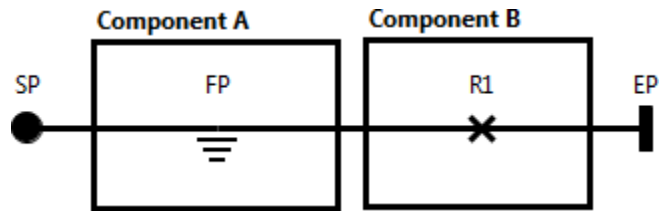
(b) Target SD

Figure 49: Mapping of timers

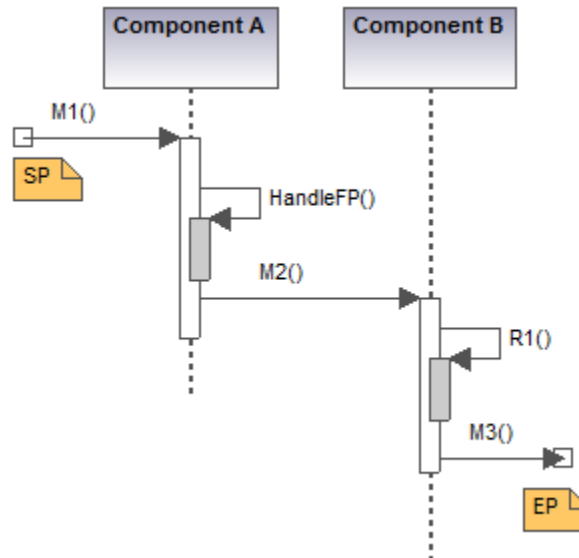
5.1.6 Failure Point

A UCM path indicates possible occurrence of erroneous or exceptional situations at failure points. UCMs do not specify how the exceptional conditional can be handled. This can be represented in SDs by self messages which handle the exception. The message should be labeled 'Handle' followed by the failure point's name.

Figure 50 illustrates the mapping of a UCM containing a failure point. In the source UCM (Figure 50(a)), the path emerging from *SP* contains a failure point *FP* in *Component A*. *FP* is translated to a self message *HandleFP()* on lifeline *Component A* in the target SD (Figure 50(b)). This indicates that lifeline *Component A* handles the erroneous situation by invoking internal message *HandleFP()* before transferring control to lifeline *Component B*.



(a) Source UCM



(b) Target SD

Figure 50: Mapping of failure points

5.1.7 Nested Components

In UCMs, a component may be composed of one or more smaller components. The UCM paths inside the nested components are represented as separate SDs, which are referenced from the main SD through InteractionUses.

Figure 51 illustrates the mapping of a UCM containing nested components. The source UCM (Figure 51(a)) contains *Component A*, which is composed of two Components, *Component A1* and *Component A2*. Their behavior is depicted in a separate SD (Figure 51 (c)), which is referenced from the target SD (Figure 51(b)) through

InteractionUse *Internal*. The InteractionUse is placed on lifeline *Component A* in order to indicate its internal structure and to preserve the flow of control depicted in the UCM.

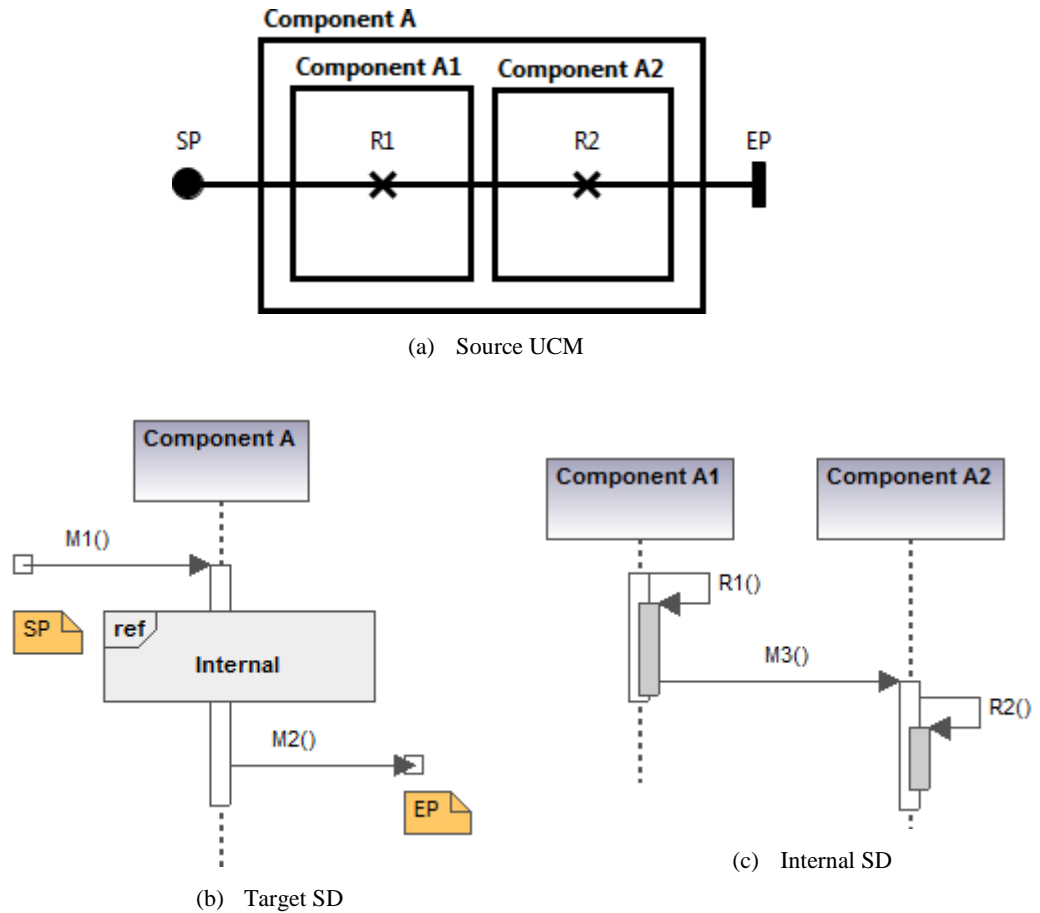


Figure 51: Mapping of nested components

5.1.8 Stub

A UCM can be refactored into smaller UCMs using stubs. Similarly, a complex SD can be modularized using InteractionUse(s). Therefore, stubs can be represented in SDs using InteractionUses.

Figure 52 illustrates the mapping of a UCM containing a stub. The source UCM (Figure 52(a)) contains a stub *ST*, whose contents are shown in Figure 52(b). *ST* is bound to *Component A*; this implies that its enclosing responsibility, *R1*, is also bound to *Component A*. The flow inside the stub is represented in a separate SD (Figure 52(d)), which is referenced from the target SD (Figure 52(c)) through InteractionUse *ST*.

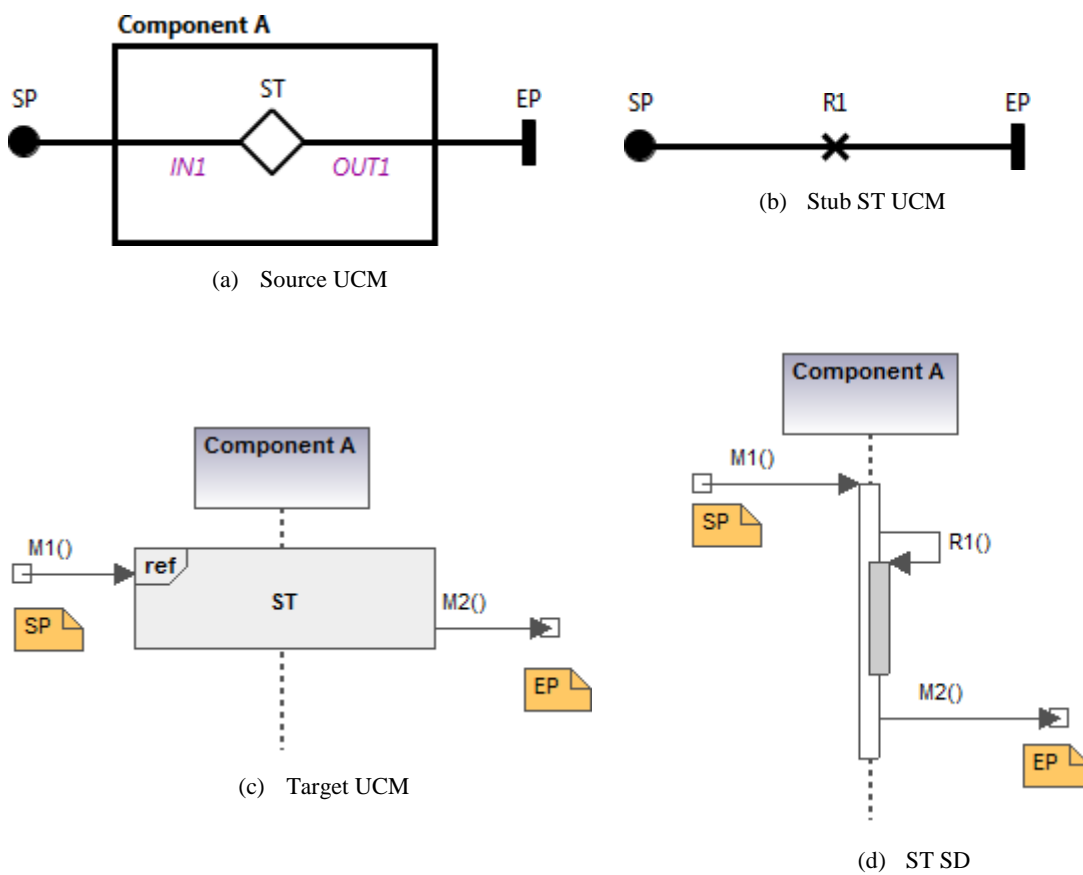


Figure 52: Mapping of stubs

5.1.9 Dynamic Stubs

A dynamic stub represents multiple stubs on a UCM path; one of which executes depending on its guard condition. They can be depicted in SDs using the alt fragment. For each stub in a dynamic stub, an operand is included in the fragment and its guard condition. The content of each stub is shown in a different SD and referenced from the target SD through InteractionUses.

Figure 53 illustrates the mapping of a UCM containing a dynamic stub. The source UCM (Figure 53(a)) contains a dynamic stub *DS*. The contents of *DS* include stubs *ST1* (Figure 53(b)) and *ST2* (Figure 53(c)). *DS* is bound to *Component A*; this implies that its enclosing responsibilities, *R1* and *R2*, are also bound to *Component A*. The guard conditions of *DS* are *[cond]* and *[!cond]* (not shown on figure). *ST1* executes when *[cond]* is satisfied, whereas *ST2* executes when *[!cond]* is satisfied. The flow inside each stub is represented in separate SDs (Figure 53(e) and (Figure 53(f)), which are referenced from the target SD (Figure 53(d)) through InteractionUses *ST1* and *ST2*.

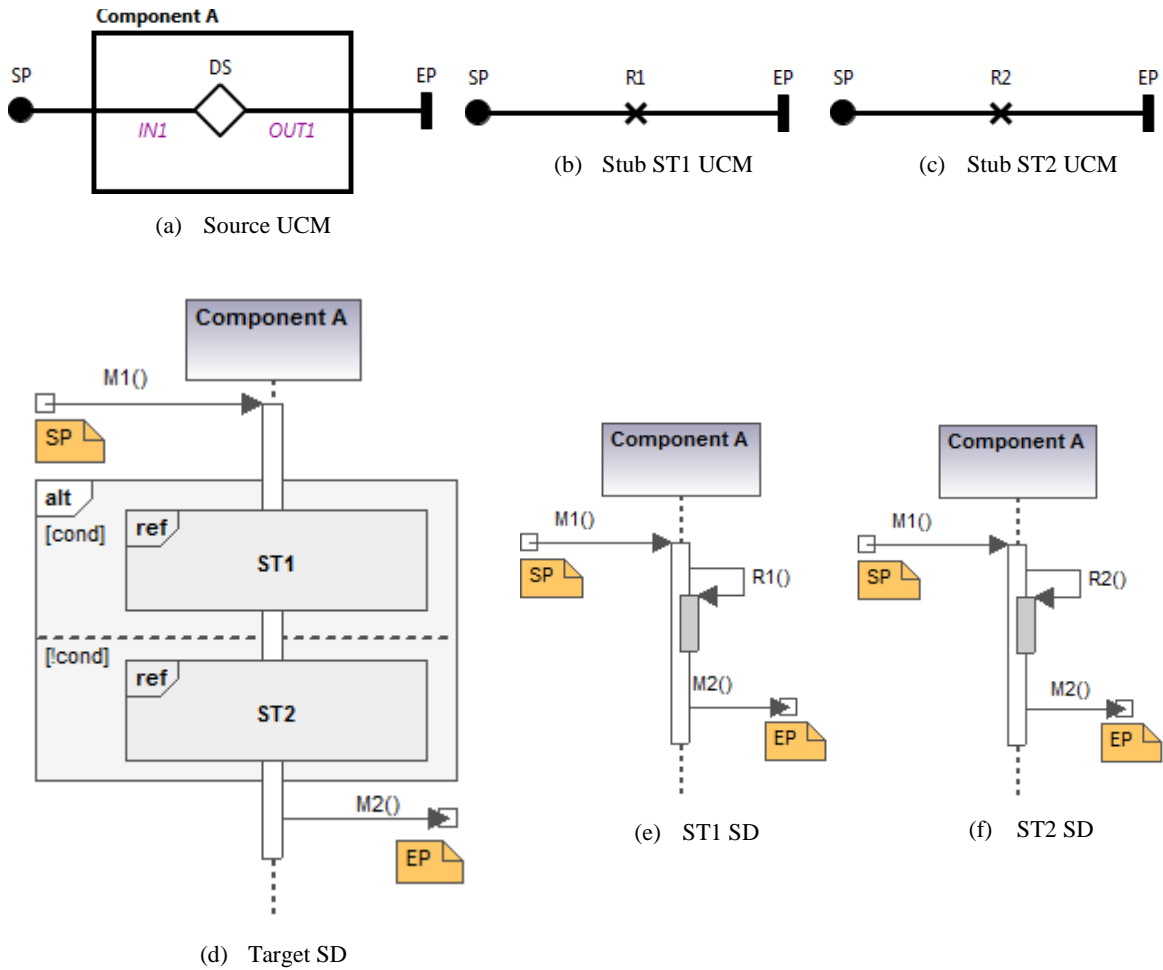


Figure 53: Mapping of dynamic stubs

5.2 Transformation Rules

In a model-driven software development approach models are automatically derived from one another to ensure consistency. In this section, we present rules for automated transformation of UCMs to UML 2 SDs. The proposed mapping was implemented using the Atlas Transformation Language (ATL), a model transformation language. The presented rules will map UCM components to UML lifelines, UCM

responsibilities to UML internal messages, and path transition between components to synchronous messages between lifelines. The mapping of alternate paths cannot be automated due to a severe limitation in the SD metamodel. The *CombinedFragment* metaclass which represents SD fragments is not associated with the *Message* metaclass which represents SD messages. Hence, the presented model transformation is semi-automated; it requires the designer to manually group the SD messages into appropriate fragments based on the proposed mapping. The ATL mapping rules are presented in Listing 26. The rules were written against the UCM metamodel and UML 2 metamodel shown in Appendix A and Appendix B, respectively.

```

entrypoint rule Main() {
  using {
    startPoint: UCM!"ucm::map::StartPoint" = thisModule.rootUCM.startPoint();
    endPoint: UCM!"ucm::map::EndPoint" = thisModule.rootUCM.endPoint();
  }
  do {
    thisModule.OrderResps(startPoint);
    thisModule.CreateLifelines();
    thisModule.CreateStartGate(startPoint);
    thisModule.CreateMessages();
    thisModule.CreateEndGate(endPoint);
  }
}

rule CreateModel {
  from d: UCM!"urn:URNspec"
  to p: UML!Model (
    packagedElement <- package
  ),
  package: UML!Package (
    packagedElement <- collaboration
  ),
  collaboration: UML!Collaboration (
    ownedBehavior <- interaction
  ),
  interaction: UML!Interaction (
    lifeline <- thisModule.lifeLines,
    fragment <- thisModule.fragments,
    message <- thisModule.messages,
    formalGate <- thisModule.formalGates
  )
}

rule CreateStartGate(sp: UCM!"ucm::map::StartPoint") {
  to g: UML!Gate (
    name <- sp.name
  )
  do {
    thisModule.formalGates <- thisModule.formalGates->including(g);
  }
}

rule CreateEndGate(ep: UCM!"ucm::map::EndPoint") {
  to g: UML!Gate (
    name <- ep.name,
  )
  do {
    thisModule.formalGates <- thisModule.formalGates->including(g);
  }
}

rule OrderResps(node: UCM!"ucm::map::PathNode") {
  do {
    if(not node.isSuccessorEndPoint() and node.isSuccessorResp()) {
      thisModule.respList <- thisModule.respList->including(node.successor().respDef);
      thisModule.OrderResps(node.successor());
    }
  }
}

rule CreateLifelines() {
  do {
    for(contRef in thisModule.rootUCM.contRefs) {
      thisModule.lifeLines <- thisModule.lifeLines
      ->including(thisModule.CreateLifeline(contRef));
    }
  }
}

```

```

rule CreateLifeline(compRef: UCM!"ucm::map::ComponentRef") {
  to l: UML!Lifeline (
    name <- compRef.getLifelineName(),
  )
  do {
    thisModule.compRefMap <- thisModule.compRefMap->including(compRef, l);
    l;
  }
}

rule CreateMessages() {
  do {
    for(resp in thisModule.respList) {
      thisModule.messages <- thisModule.messages
      ->including(thisModule.CreateMessage(resp));
    }
  }
}

rule CreateMessage(resp: UCM!"urncore::Responsibility") {
  to m: UML!Message (
    messageSort <- 'synchCall',
    sendEvent <- thisModule.respMessageStartMap->get(resp),
    receiveEvent <- thisModule.respMessageEndMap->get(resp),
    name <- resp.name
  )
  do {
    m;
  }
}

```

Listing 26: UCM to UML 2 SD ATL mapping rules

5.3 Case Study

5.3.1 Source Model

In this section, we apply the proposed transformation on a case study that pertains to the UCM (Figure 54) of an Elevator Control System (ECS), and is taken, with permission, from [8]. This UCM is a refined version of the UCM used in Chapter 4; it contains additional components and paths. The ECS contains a set of components that interact with each other to provide the required functionality. The *Service Personnel Interface* (SPI) allows turning on of the ECS. Each elevator's states (*stationary* and *moving*) are controlled by the *Elevator Control* (EC) component. When the ECS is turned on through the SPI, all EC components are in the stationary state. The *Elevator Manager*

(EM) component determines whether an elevator request can be granted or not. If it can be granted, it signals the elevator's EC component to start moving. The *Status and Planner* (SP) component maintains and manages the list of all elevators that are being used. The *Elevator* component controls an elevator's motor and door movement. An *Arrival Sensor* component on each elevator informs the EC that it is about to approach a floor. The *Status and Planner* (SP) also determines whether the floor being approached by an elevator is the requested one or not. If it is the requested one, the elevator stops and its door opens. The SP component removes the elevator from the list of elevators that are being used. The elevator goes back to stationary state and waits for requests from its EM. User can make requests from outside the elevator or from inside. Requests made from outside are *up* and *down*; *above* and *below* are the requests made from inside. Outside requests are sent to the *Scheduler* component, which selects an elevator to satisfy the request, and forwards it to the elevator's respective EM. Inside requests are directly sent to an elevator's EM. The UCM of the ECS is shown in Figure 54.

5.3.2 Scenario Extraction

Before we apply the proposed mapping rules, individual scenarios must be extracted from the ECS UCM. An individual scenario represents complete the execution of a UCM path. The benefit of individual scenarios is that they allow the validation of requirements, and ease the transition from requirements to design [7]. The possible individual scenarios that can be extracted from the ECS UCM are as show in Table 6.

Table 6: All possible scenarios in ECS UCM

Scenario	Sequence
S1	<i>up</i>
S2	<i>down</i>
S3	<i>above</i>
S4	<i>below</i>
S5	<i>approaching floor, moving</i>
S6	<i>at floor, floor input, select elevator, add to list, [on list], already on list</i>
S7	<i>in elevator, elevator input, add to list, [on list], already on list</i>
S8	<i>at floor, floor input, select elevator, add to list, [!on list]</i>
S9	<i>in elevator, elevator input, add to list, [!on list]</i>
S10	<i>switch on, stationary, decide on direction, close door, [up], motor up, moving, [requested], motor stop, door open, remove from list, <at requested floor, door closing delay>, stationary</i>
S11	<i>switch on, stationary, decide on direction, close door, [down], motor down, moving, [requested], motor stop, door open, remove from list, <at requested floor, door closing delay>, stationary</i>
S12	<i>switch on, stationary, decide on direction, close door, [up], motor up, moving, [!requested]*, [requested], motor stop, door open, remove from list, <at requested floor, door closing delay>, stationary</i>
S13	<i>switch on, stationary, decide on direction, close door, [down], motor down, moving, [!requested]*, [requested], motor stop, door open, remove from list, <at requested floor, door closing delay>, stationary</i>
* represents a loop	

Scenario S1 represents the case in which a user chose to go up, whereas scenario S2 represents the case in which the user chose to go down, using the panel outside the elevator. Scenario S3 represents the case in which a user choose to go up, whereas scenario S4 represents the case in which a user choose to go down, using the panel inside the elevator.

Scenario S5 represents the case in which the elevator is signaled that the approaching floor is the destination floor. Scenarios S6 and S7 represent the case in which the elevator selected for satisfying the user's request is busy. On the contrary, scenarios S8 and S9 represent the case in which the selected elevator is free to satisfy the user's request.

Scenarios S10 and S11 represent the case in which the destination floor is exactly the next floor (above or below). On the contrary, scenarios S12 and S13 represent the case in which the destination floor is more than one floor apart (above or below). S12 and S13 loop at guard condition *[! requested]* until the floor that is being approached by the elevator is the destination floor. It should be noted that the source UCM contains an infinite loop that starts and ends at waiting point *stationary*. Therefore, scenarios S10, S11, S12, and S13 end at the second occurrence of *stationary*.

The paths of scenarios S1, S5, S8 and S12 are combined to produce the UCM in Figure 55. This combined UCM will be used to demonstrate the mapping of the ECS UCM to UML 2 SD notation.

The target SD of S1 (Figure 56) contains a state invariant *up*, on lifeline *User*, which represents the user's destination selection.



Figure 56: Mapping of scenario S1 to SD notation

Scenario S5

On the path of scenario S5, start point *approaching floor*, which is bounded to component *Arrival Sensor*, signals the elevator that a floor is about to be approached. S5 ends after transferring control to the *Elevator Control* component. The target SD of S5 (Figure 57) contains state invariant *approaching floor*, on lifeline *Arrival Sensor*, and message *M1()* passed to lifeline *Elevator Control*.

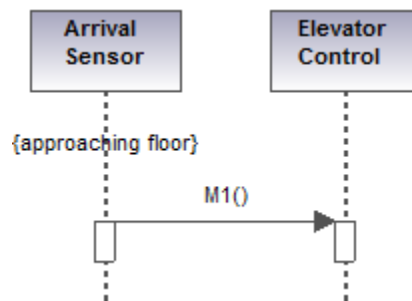


Figure 57: Mapping of scenario S5 to SD notation

Scenario S8

The path of scenario S8 begins at start point *at floor*, which is bounded to component *User*, and then pauses at waiting point *floor input* for the user to select *up* as the destination, i.e. completion of scenario S1. The target SD of S8 (Figure 58) begins at state invariant *at floor* on lifeline *User*, and then waits, at state invariant *floor input*, for scenario S1 to complete its execution. The SD of S1 is referenced using InteractionUse *S1*.

After selection of the *up* destination, control is transferred to the *Scheduler* component. This transfer is depicted in target SD by the invocation of message *M2()* on lifeline *Scheduler*. The designer must rename this message appropriately during refinement of this SD. In order to satisfy the user's request, *Scheduler* chooses a free elevator by performing responsibility *select elevator*, which is mapped in the target SD as self message *selectElevator()*, which invoked by lifeline *Scheduler*.

After an elevator is selected, control is transferred to the *Elevator Manager* component, which further transfers control to the *Status and Planner* component. This successive transfer is shown in the target SD as a sequence of messages, *M3()* and *M4()*. The destination of *M3()* is lifeline *Elevator Manager*, whereas that of *M4()* is lifeline *Status and Planner*.

Status and Planner now adds the selected elevator to the list of elevators in operation by performing responsibility *add to list*, and then transfers control back to the *Elevator Manager* component. In the target SD, lifeline *Status and Planner* invokes self

message *addToList()*, and then passes message *M5()* to lifeline *Elevator Manager*. *Elevator Manager* now forwards control to the *Elevator Control* component, which then ends S8. In the target SD, lifeline *Elevator Manager* passes message *M6()* to lifeline *Elevator Control*, thus ending the SD.

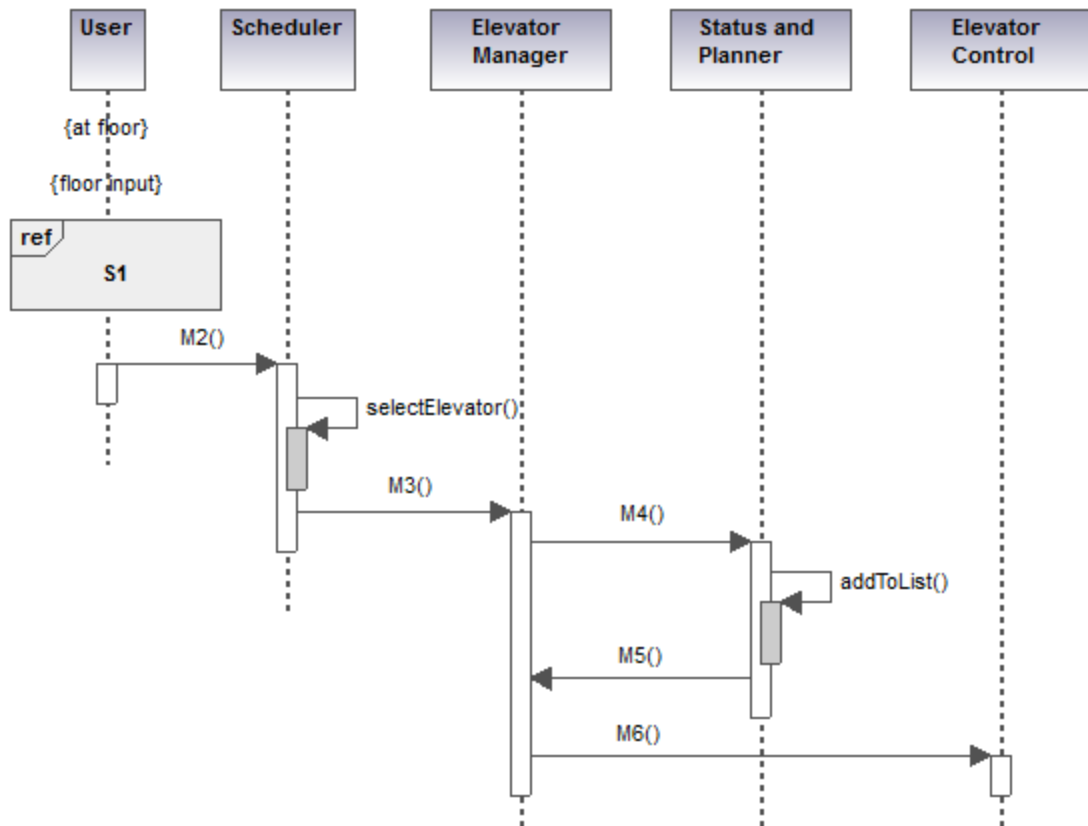


Figure 58: Mapping of scenario S8 to SD notation

Scenario S12

The path of scenario S12 begins at start point *switch on*, which is bounded to component *Service Personnel Interface*, and then transfers control to component *Elevator Control*. Now, the path pauses, at waiting point *stationary*, for a free elevator to be selected, i.e. completion of scenario S8. The target SD of S12 (Figure 59) begins at state invariant *switch on* on lifeline *Service Personnel Interface*, and then passes message *M7()* to lifeline *Elevator Control*, which waits at state invariant *stationary* for S8 to complete its execution. The SD of S8 is referenced using *InteractionUse S8*.

After the completion of scenario S8, responsibility *decide on direction* is performed in order to determine whether the elevator should go up or down. This is represented on the target SD as self message *decideOnDirection()*, which is invoked by lifeline *Elevator Control*. In S12, the outcome of *decide on direction* will always be up because the user choose to go up.

After the direction of the elevator is determined, control is transferred to the *Elevator* component, which shuts the door of the elevator, and then moves the elevator upwards by performing responsibilities *close door* and *motor up*, respectively. In the target SD, lifeline *Elevator Control* passes message *M8()* to lifeline *Elevator*, which invokes self messages *closeDoor()* and *motorUp()*, in sequence.

After the elevator begins its ascent towards the destination floor, control returns to the *Elevator Control* component, which pauses, at waiting point *moving*, for a floor to be approached, i.e. completion of scenario S5. In the target SD, lifeline *Elevator* passes message *M9()* to lifeline *Elevator Control*, which pauses, at state invariant *moving*, for InteractionUse *S5* to finish its execution.

As the elevator is about to approach a floor, control gets transferred to the *Status and Planner* component, which determines whether the floor is the requested one or not. If guard condition *[requested]* is satisfied, control is transferred to the *Elevator* component; otherwise, control returns back to the *Elevator Control* component. In scenario S12, *[!requested]* holds an indefinite number of time before *[requested]* is satisfied. This is shown in the target SD by a loop fragment having guard condition *[!requested]*. It should be noted that this mapping is the alternate mapping of UCM loops

(see Section 5.1.2.3). Messages *M10()* and *M11()* represent the repeated back and forth transfer of control between lifelines *Elevator Control* and *Status and Planner*. When the guard condition of the loop fragment does not hold, lifeline *Status and Planner* passes message *M12()* to lifeline *Elevator*.

After the elevator reaches its destination, the *Elevator* component halts the elevator, and then opens its door by performing responsibilities *motor stop* and *door open*, respectively. In the target SD, lifeline *Elevator* invokes self messages *motorStop()* and *doorOpen()*, in sequence.

The elevator must now be removed from the list of elevators busy elevators. This is achieved by transferring control back to the *Status and Planner* component, which performs responsibility *remove from list*. In the target SD, lifeline *Elevator* passes message *M13()* to lifeline *Status and Planner*, which invokes self message *removeFromList()*.

The elevator must now be able to receive another request after its door shuts. This is achieved by transferring control back to the *Elevator Control* component, which halts S12 at timer *door closing delay* for an arbitrary amount of time. Simultaneously, control is also transferred to the *User* component to indicate that the user has reached his destination. This simultaneous transfer is achieved by AND-fork AF, which is represented as a par fragment in target SD. The par fragment contains two operators; the first passes message *M14()* to lifeline *Elevator Control*, whereas the second passes message *M15()* to lifeline *User*. The halting of S12 is represented by state invariant *door closing delay* on lifeline *Elevator Control*. After an arbitrary amount of time the *Elevator*

Control component is ready to receive another request while halting at waiting point *stationary*. This is represented in the target SD as state invariant *stationary* on lifeline *Elevator Control*. The invocation of *M15()* triggers state invariant *at requested floor* on lifeline *User*.

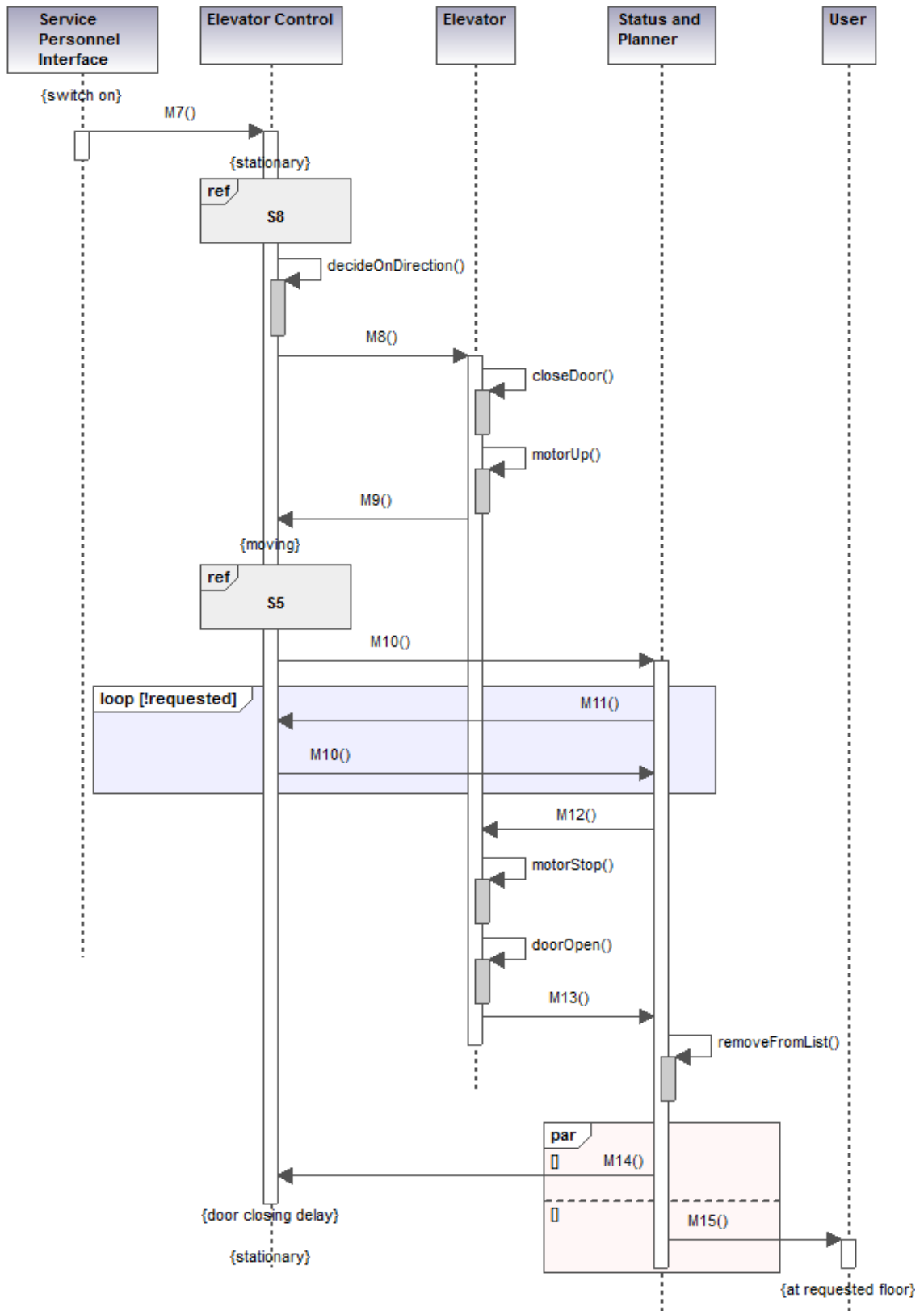


Figure 59: Mapping of scenario S12 to SD notation

CHAPTER 6

A MUTATION FRAMEWORK FOR MODEL TRANSFORMATIONS

The widespread interest in testing model transformation programs provides the major motivation for this chapter. This chapter, in particular, focuses on investigating the applicability of fault based testing to model transformations. This chapter serves the following purposes:

- It proposes a suite of mutation operators for the Atlas Transformation Language (ATL), so that model transformation developers can gain the benefits of mutation testing.
- It presents a prototype tool, *MuATL*, for automatic generation of ATL mutants.

The remainder of this chapter is organized as follows. Our proposed ATL mutation testing approach is presented in Section 6.1. Section 6.2 introduces a suite of 10 mutation operators for the ATL transformation language. An analysis of the proposed mutation operators follows in Section 6.3. An automated tool for ATL mutant generation is described in Section 6.4. In Section 6.5, we apply the defined mutation operators on the UCM to UML 2 AD model transformation defined in Chapter 4. A discussion follows in Section 6.6.

6.1 ATL Mutation Testing Approach

Mutation testing is a well-established fault based testing technique, in which faults are seeded into a syntactically correct program, in order to determine the efficiency of a test suite. Mutation testing has been successfully applied to various areas and languages: programming languages (e.g., Fortran, Ada, C, Java), integration testing (e.g., interface mutation), design models (e.g., Finite State Machines, petri nets, state-charts), web services, etc. For a comprehensive survey on the development of mutation testing, the reader is invited to consult [84].

An ATL *mutation operator* defines how a particular ATL artifact will be changed in order to seed a fault. Application of a mutation operator results in a defective ATL program, which is known as a *mutant* ATL program. If a mutant is syntactically incorrect, it is considered as an *invalid* mutant.

Figure 60 illustrates the general mutation process for ATL. An ATL test suite consists of a synthesis of a number of input models as test cases. The original ATL program and the generated mutants run on the test cases, and the results are compared using an oracle. Defining a test oracle for model transformations is a challenging task [26][135]. Indeed, the number of constraints to define can be very large to cover all transformation possibilities [26].

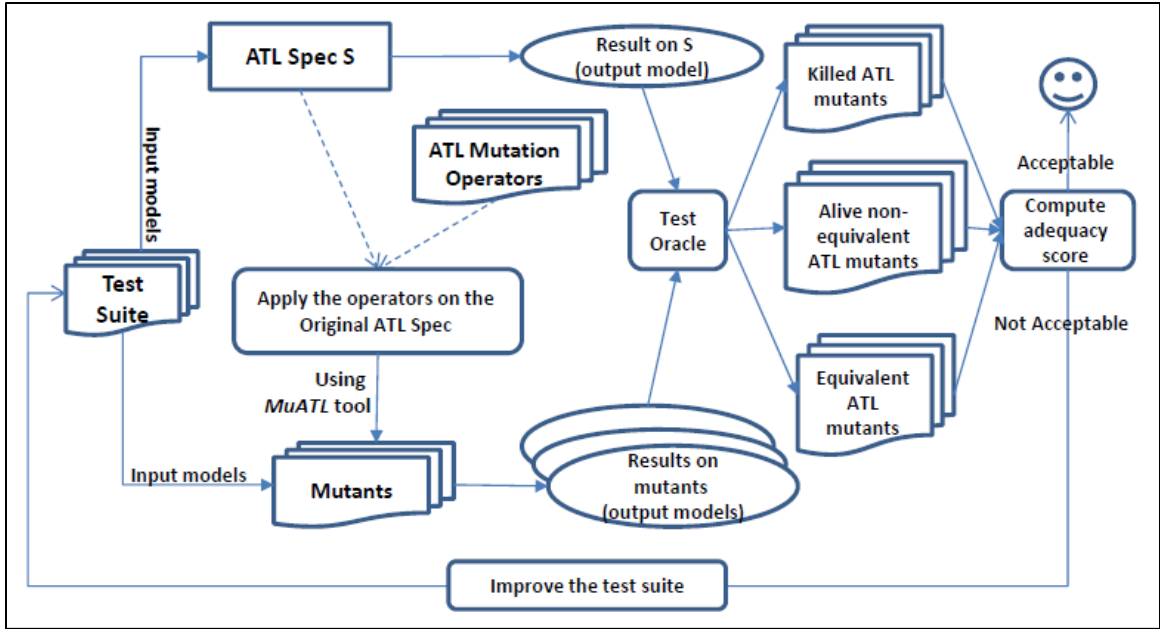


Figure 60: ATL mutation process

A given test case, part of the test suite, is said to *kill* a mutant if the output model produced by the mutant is different from that of the original ATL specification. Hence, the test case is good enough to detect the change between the original and the mutant ATL program. A test case cannot distinguish between a mutant and the original ATL program if both produce the same output model for the same input model(s). If a mutant is not killed (called *alive*) by a test suite, this usually means that the test suite is not adequate. However, it may also be that the mutant keeps the program's semantics unchanged; thus, cannot be detected by any test case. Such mutants are called *equivalent* mutants. Equivalent mutant detection is, in general, one of biggest obstacles for practical usage of mutation testing. The effort needed to check if mutants are equivalent or not, can be very high even for small programs [84].

ATL Mutants are generated automatically using our prototype tool *MuATL* (see Section 6.4). The execution of the test suite and the oracle function are performed manually. The automation of such activities is out of the scope of this chapter.

The effectiveness of a test suite TS_{eff} is determined by running it on all mutants, and computing the ratio of killed mutants to total number of non-equivalent mutants. TS_{eff} is given by the following equation:

$$TS_{eff} = \frac{M_k}{M_t - M_e} \quad (1)$$

where M_k is the number of killed ATL mutants, M_t is the total number of generated ATL mutants, and M_e is the number of ATL equivalent mutants. If the score is not acceptable, the test suite should be improved by adding additional test cases and/or modifying the existing ones.

6.2 ATL Mutation Operators

In this section, mutation operators are defined for ATL, and code samples are shown to demonstrate their usage. The number of possible mutants that can be generated for certain operators is specified. The consequences of applying the mutation operators are also described. For a brief overview on ATL, the reader may consult Section 1.6.

6.2.1 Matched to Lazy (M2L)

ATL gives developers the flexibility to define model transformations in both declarative and imperative styles. Matched rules are declarative rules that are implicitly called by the ATL virtual machine at runtime. The M2L operator converts a matched rule to a lazy rule (which is an imperative rule). The consequence of applying the M2L operator is that a mutant rule will never be executed, since lazy rules must be explicitly invoked. The number of M2L mutants that can be created for given ATL module is equal to the number of matched rules it contains.

An example of a mutation performed by applying the M2L operator is shown in Table 7. The M2L operator prepends the rule *AtoB* by the lazy modifier in the mutant rule *AtoB'*.

Table 7: Example of a M2L mutation

Original	Mutant
<pre>rule AtoB { from s : A to t: B (.....) }</pre>	<pre>lazy rule AtoB' { from s : A to t: B (.....) }</pre>

6.2.2 Lazy to Matched (L2M)

The L2M operator does the opposite of the M2L operator; it converts a lazy rule into a matched rule. Matched rules cannot be explicitly invoked; therefore, a runtime failure will occur when a L2M mutant rule is called. However, a L2M mutation cannot be detected if the mutant rule is not invoked during an execution. The number of L2M

mutants that can be created for a given ATL module is equal to the number of lazy rules it contains.

An example of a mutation performed by applying the L2M operator is shown in Table 8. The L2M operator deletes the *lazy* modifier of rule *AtoB* in the mutant rule *AtoB'*.

Table 8: Example of a L2M mutation

Original	Mutant
<pre> lazy rule AtoB { from s : A to t: B (.....) } </pre>	<pre> rule AtoB' { from s : A to t: B (.....) } </pre>

6.2.3 Delete Attribute Mapping (DAM)

Attribute mapping(s) in an ATL rule define how a source object will be transformed into a target object. The DAM operator deletes an attribute mapping from the definition of a particular rule. It is based on the CACD operator in [136]. The consequence of applying the DAM operator on a rule is that the attribute, whose mapping is deleted, will not participate in the transformation process, resulting in a loss of information. The DAM operator can be applied on matched, lazy and mapping called rules. The number of DAM mutants that can be created for a given rule is equal to the number of attribute mappings it contains.

An example of a mutation performed by applying the DAM operator is shown in Table 9. The DAM operator deletes the mapping of attribute *b2* in the mutant rule *AtoB'*.

Table 9: Example of a DAM mutation

Original	Mutant
<pre>rule AtoB { from s : A to t: B (b1 <- s.a1, b2 <- s.a2) }</pre>	<pre>rule AtoB' { from s : A to t: B (b1 <- s.a1) }</pre>

6.2.4 Add Attribute Mapping (AAM)

Developers may avoid transforming redundant information from a source model into a target model. In such a situation, mappings of useless attributes are not specified in the transformation rule. The AAM operator adds a useless attribute mapping from a source object to a target object in a given rule. It is based on the CACA operator in [136]. The consequence of applying the AAM operator on a rule is that unnecessary complexity is added to the output model. The number of AAM mutants that can be created for a given rule is equal to the product of the number of unmapped attributes in the source and target objects. An example of a mutation performed by applying the AAM operator is shown in Table 10. The AAM operator adds the useless mapping “*b2 <- s.a2*” in the mutant rule *AtoB'*.

Table 10: Example of an AAM mutation

Original	Mutant
<pre>rule AtoB { from s : A to t: B (b1 <- s.a1) }</pre>	<pre>rule AtoB' { from s : A to t: B (b1 <- s.a1, b2 <- s.a2) }</pre>

6.2.5 Delete Filtering Expression (DFE)

Filtering expressions constrain the input objects on which a particular rule can be applied. If a filtering statement evaluates to true for a given input object, its corresponding rule will be executed. DFE can only be applied on matched rules, as they allow filtering of input objects. The DFE operator deletes the filtering statement specified in the definition of a rule. It is based on the CFCD operator in [136]. The consequence of applying the DFE operator is that the mutant rule will be executed for incorrect objects of its source type. DFE operator may cause filtering expressions of multiple rules to evaluate to true for one source instance. In this case, a runtime failure will occur. The number of DFE mutants that can be created for a given ATL module is equal to the number of matched rules that contain a filtering expression.

An example of a mutation performed by applying the DFE operator is shown in Table 11. The DFE operator removes the filtering expression $s.a1 > 0$ in mutant rule *AtoB'*.

Table 11: Example of a DFE mutation

Original	Mutant
<pre>rule AtoB { from s : A (s.a1 > 0) to t: B (b1 <- s.a1, b2 <- s.a2) }</pre>	<pre>rule AtoB' { from s : A to t: B (b1 <- s.a1, b2 <- s.a2) }</pre>

6.2.6 Add Filtering Expression (AFE)

Based on the CFCA operator in [136], we define the AFE operator which performs the opposite of the DFE operator. It adds an unnecessary filtering expression to a matched rule. The consequence of applying the AFE operator is that some objects of the input model will not participate in the transformation process; thus, resulting in a loss of information. In order to apply the AFE operator on a rule, the source object must have at least one attribute. If this condition is satisfied, numerous AFE mutants can be created for a given matched rule. A mutant generation tool can constrain the possible number of AFE mutants. Similar to the DFE operator, the AFE operator can also cause a runtime failure.

An example of a mutation performed by applying the AFE operator is shown in Table 12. The AFE operator adds the filtering expression $s.a1 > 0$ in mutant rule $AtoB'$. $a1$ is a scalar attribute in source object s .

Table 12: Example of a AFE mutation

Original	Mutant
<pre>rule AtoB { from s : A to t: B (b1 <- s.a1, b2 <- s.a2) }</pre>	<pre>rule AtoB' { from s : A (s.a1 > 0) to t: B (b1 <- s.a1, b2 <- s.a2) }</pre>

6.2.7 Change Source Type (CST)

ATL rules define mappings from source objects to target objects. The CST operator changes the source type of a given rule. It can be applied on matched and lazy rules. The consequence of applying the CST operator is that incorrect transformations may be performed. Indeed, the application of the CST operator on a rule will cause a runtime failure if the new source type does not contain the attributes which are specified to be mapped, or if multiple rules are associated with the new source type. The number of CST mutants that can be created for a given rule is equal to the number of classes in the source metamodel that participate in the transformation minus one.

An example of a mutation performed by applying the CST operator is shown in Table 13. The source type of rule *AtoB* is changed from *A* to *C* in the mutant rule *AtoB'*.

Table 13: Example of a CST mutation

Original	Mutant
<pre>rule AtoB { from s : A to t: B (.....) }</pre>	<pre>rule AtoB' { from s : C to t: B (.....) }</pre>

6.2.8 Change Target Type (CTT)

The CTT operator changes the target type of a given rule. It can be applied on matched, lazy, and mapping called rules. The consequence of applying the CTT operator is that the objects in the input model will be transformed into objects of incorrect type in the output model. Application of the CTT operator on a rule will cause a runtime exception if the new target type does not contain the attributes which are specified to be mapped. The number of CTT mutants that can be created for a given rule is equal to the number of classes in the target metamodel that participate in the transformation minus one.

An example of a mutation performed by applying the CTT operator is shown in Table 14. The target type of rule *AtoB* is changed to *C* in the mutant rule *AtoB'*.

Table 14: Example of a CTT mutation

Original	Mutant
<pre>rule AtoB { from s : A to t: B (.....) }</pre>	<pre>rule AtoB' { from s : A to t: C (.....) }</pre>

6.2.9 Change Execution Mode (CEM)

ATL modules can execute in two modes, *default* and *refining*. Default mode is the default execution mode of ATL transformations and it is specified by the *from* keyword. The refining mode allows developer to specify rules only for those objects that need to be transformed; remaining objects will be implicitly copied into the output model. It should be added that refining mode applies only when the source and target models conform to the same metamodel. We define the CEM operator which switches the execution mode of an ATL module from default to refining mode. If a module contains imperative code, which is not allowed in refining mode, application of the CEM operator will result in an invalid (i.e., syntactically incorrect) mutant. The consequence of the CEM mutation is that useless objects may be copied into the output model. A single CEM mutant can be created for a given module.

An example of a mutation performed by applying the CEM operator is shown in Table 15. The CEM operator changes the execution mode of module *A* to refining mode in the mutant module *A'*.

Table 15: Example of a CEM mutation

Original	Mutant
<code>module A; create OUT : UML from IN : UML;</code>	<code>module A'; create OUT : UML refining IN : UML;</code>

6.2.10 Delete Return Statement (DRS)

The last statement of a *do* block in a mapping called rule must return the target object. It is optional to specify a return statement in the *do* block of matched and lazy rules. The DRS mutation operator deletes the return statement of a *do* block. The number of DRS mutants that can be created for a given rule is equal to the number of return statements in the *do* block; a *do* block may use conditional blocks to have several return statements.

An example of a mutation performed by applying the DRS operator is shown in Table 16. The DRS operator deletes the return statement “*t*;” of the *do* block of rule *AtoB* in mutant rule *AtoB'*.

Table 16: Example of a DRS mutation

Original	Mutant
<pre> lazy rule AtoB { from s : A to t: B (.....) do { t; } } </pre>	<pre> lazy rule AtoB' { from s : A to t: B (.....) do { } } </pre>

6.3 ATL Mutation Operator Analysis

Table 17 summarizes the proposed ATL mutation operators. For each operator, we specify the ATL execution mode in which it is applicable, the type of rules on which the operator can be applied, and the number of expected mutants per rule.

It is worth noting that the Table 17 does not contain operators that can be applied to non-mapping called rules. Non mapping-called rules are similar to void functions in other traditional programming languages. So, mutation operators from other traditional programming languages that pertain to functions can be applied on non-mapping called rules. Covering such mutants is out of the scope of this chapter.

Table 17: Summary of ATL mutation operators

Operator	Execution Mode	Rules	Expected number of generated mutants per rule
M2L	Default	Matched	1
L2M	Default	Lazy	1
DAM	Default, Refining	Matched, Lazy, Mapping Called	Number of attribute mappings
AAM	Default, Refining	Matched, Lazy, Mapping Called	Product of number of unmapped attributes in the source and target objects
DFE	Default, Refining	Matched	1
AFE	Default, Refining	Matched	Many possibilities
CST	Default, Refining	Matched, Lazy	Number of classes in the source metamodel that participate in the transformation minus one
CTT	Default, Refining	Matched, Lazy, Mapping Called	Number of classes in the target metamodel that participate in the transformation minus one
CEM	Default	-	1
DRS	Default	Matched*, Lazy*, Mapping Called	Number of return statements
* If their <i>do</i> block contains a return statement			

6.3.1 Number of Generated ATL Mutants

In what follows, we provide the general formulas to compute the maximum number of mutants relative to the defined operators, when applied to a complete ATL module.

Let R_m , R_l , and R_{mc} be the number of matched rules, the number of lazy rules, and the number of mapping called rules, respectively, in a given ATL module. The maximum numbers of CST mutants that can be generated for an ATL module is given by the following equation:

$$M_{CST} = (sc - 1)(R_m + R_l) \quad (2)$$

where sc represents the number of source metaclasses that participate in the model transformation.

The maximum numbers of CTT mutants that can be generated for an ATL module is given by the following equation:

$$M_{CTT} = (tc - 1)(R_m + R_l + R_{mc}) \quad (3)$$

where tc represents the number of target metaclasses that participate in the model transformation.

The maximum number of DAM mutants that can be generated for an ATL module is:

$$M_{DAM} = \sum_{i=0}^{R_m+R_l+R_{mc}} am_i \quad (4)$$

where am_i is the number of attribute mappings in a given rule i .

The maximum number of AAM mutants that can be generated for an ATL module is:

$$M_{AAM} = \sum_{i=0}^{R_m+R_l+R_{mc}} usa_i * uta_i \quad (5)$$

where usa_i and uta_i denote the number of unmapped attributes for the source and target metaclasses, respectively, in a give rule i .

The maximum number of DRS mutants that can be generated for an ATL module is:

$$M_{DRS} = \sum_{i=0}^{R_m+R_l+R_{mc}} r_i \quad (6)$$

where r_i is the number of return statements in the *do* block of a given rule i .

6.3.2 Equivalent ATL Mutants

Applying the CST operator on lazy rules will always produce equivalent mutants. Indeed, the incorrect source type of a mutant lazy rule does not affect its execution. The source type of a lazy rule is decided, at runtime, by the actual parameter passed into it. The type of the actual parameter becomes the source type of the lazy rule. This implies

that the type specified in the *from* clause of a lazy rule becomes meaningless at runtime. Therefore, all produced mutants that correspond to lazy rules are considered as equivalent mutants. Hence, the CST operator is appropriate only for matched rules. This inference has been confirmed by the case study presented in Section 6.5.

6.3.3 Other Remarks

Based on the operator descriptions, and an analysis of the impact of each mutation operator, we can infer that:

- CEM operator would produce invalid mutants when applied on a module having imperative code.
- AFE operator should be applied manually, as there are numerous possible mutants.
- AAM operator would not produce any mutants for a rule, if all attributes of the source object are mapped.
- M2L operator cannot be used in refining mode since the resulting rule would become imperative, which is not allowed in refining mode.
- L2M and DRS operators are not applicable in refining mode (i.e., imperative code is not allowed in refining mode).

These remarks have been confirmed by the case study presented in Section 6.5. The DAM and AAM operators are related to the “creation” class of operators in [136]. The DFE and AFE operators are related to the “filtering” class of operators in [136]. The remaining operators M2L, L2M CST, CTT, CEM, and DRS capture the characteristics specific to ATL.

6.4 MuATL (Mutation Toolkit for ATL)

The ATL mutation operators, presented in Section 6.2, have been implemented in a prototype tool called *MuATL* (Mutation Toolkit for ATL). *MuATL*, a Microsoft .NET C# based tool, is inspired by *MuJava* (Mutation System for Java) [114].

Figure 61 illustrates the main graphical user interface of *MuATL*. The GUI is composed of two menus: (1) *Module*, and (2) *Mutation*. The user starts with loading an ATL module using the *Load* menu option. The user can select one of the mutation operators using the Mutation menu.

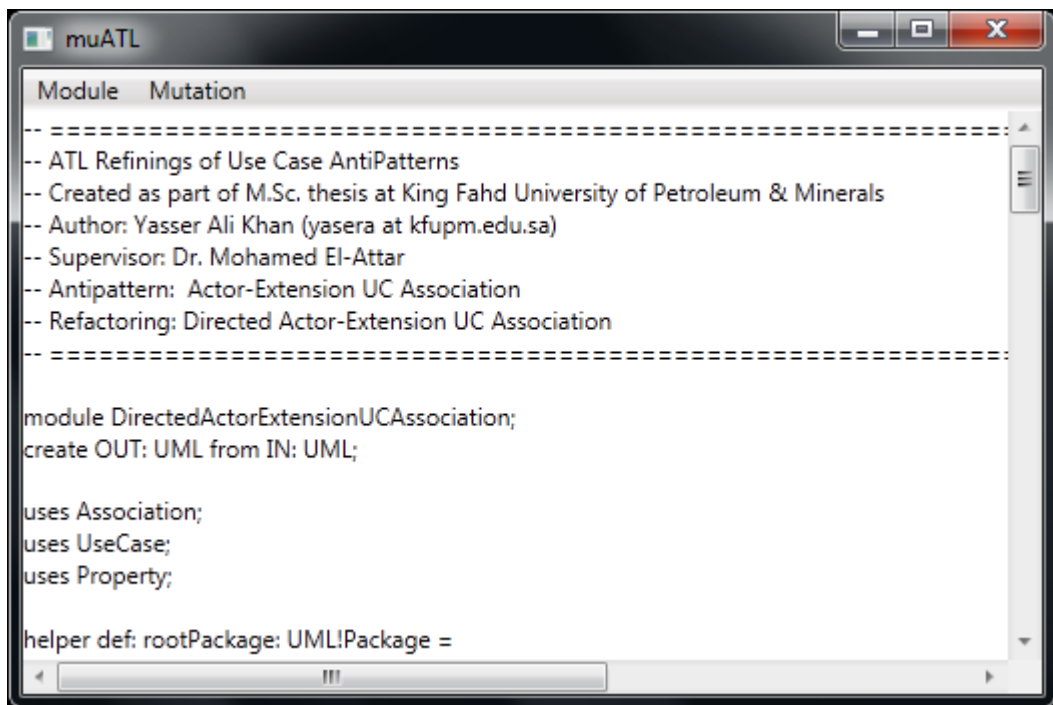


Figure 61: MuATL GUI

Mutation operators AFE, AAM, CST, and CTT require user input for mutant generation. Figure 62 illustrates the GUI where the user can select the rule, and add the corresponding filtering expression(s) for creating AFE mutants. For each filtering

expression entered, a distinct AFE mutant will be created by the tool. Similarly, AAM requires the user to enter attribute mappings for creating AAM mutants. CST and CTT require the user to enter source and target types, respectively, of the mutants. The produced mutants are stored in separate files within separate directories, each named with the operator name.

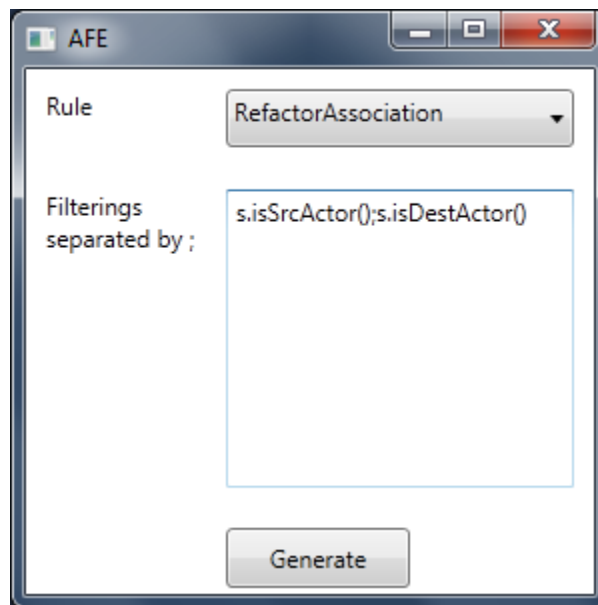


Figure 62: AFE Mutant GUI

6.5 Case Study: UCM to UML 2 AD Transformation

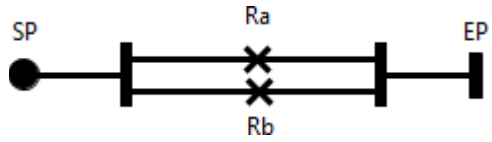
In this section, a case study is presented to show the applicability of the developed set of ATL mutation operators. Furthermore, this experiment aims at assessing the effectiveness of the proposed operators. The case study pertains to an ATL transformation program, introduced in previous work [93], which transforms the ITU-T standard [82] UCMs to UML 2 ADs.

6.5.1 Test Cases

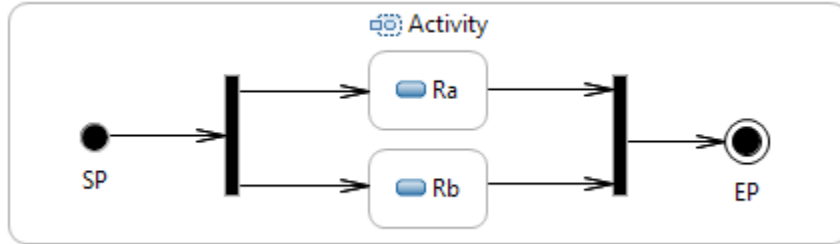
The case study is comprised of one ATL module [94] and seven test cases (see Table 18). The test cases, in Table 18, cover 16 UCM source classes and 10 AD target classes. Each test case includes the input model, the expected output model, and the actual output model. For instance, Figure 63 and Figure 64 illustrate the input model and the expected output model relative to test cases TC1 and TC2, respectively. The selected test cases satisfy the all-source-classes coverage (ASCC) criteria. This criterion ensures that all classes of the source metamodel that participate in the model transformation are covered by the test cases. It is worth noting that the ASCC criterion does not consider attribute, association, and inheritance coverage. Therefore, it is considered as a weak coverage criterion.

Table 18: Test cases of UCM to UML AD model transformation

Test Case	UCM Classes covered*	UML Classes covered*
TC1	AndFork, AndJoin, RespRef,	ForkNode, OpaqueAction
TC2	OrFork, OrJoin, RespRef	MergeNode, OpaqueAction
TC3	WaitingPlace	MergeNode
TC4	Timer, FailurePoint	OpaqueAction
TC5	EmptyPoint, DirectionArrow	-
TC6	ComponentRef	ActivityPartition
TC7	Stub	StructuredActivityNode
* All test cases cover classes URNSpec, StartPoint, EndPoint, NodeConnection		* All test cases cover classes Package, Activity, InitialNode, ActivityFinalNode, ControlFlow

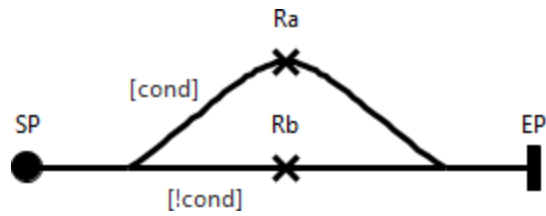


(a) UCM input model

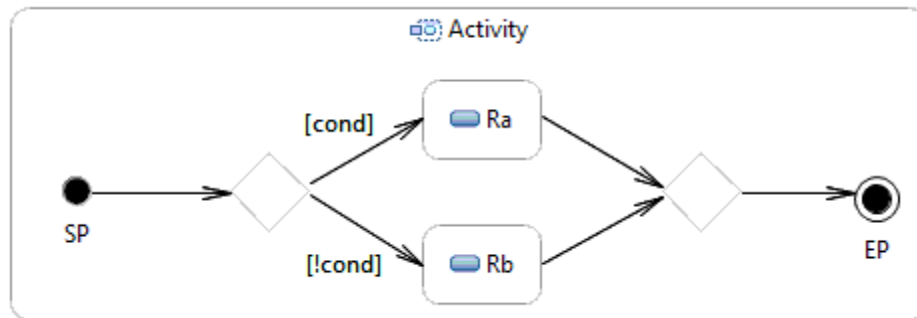


(b) AD expected output model

Figure 63: Input and expected output models of TC1



(a) UCM input model



(b) AD expected output model

Figure 64: Input and expected output models of TC2

6.5.2 Generated Mutants

The proposed mutation operators, automatically applied on the module using our prototype tool, result in 395 mutant modules. The test cases are sequentially executed, manually, on each mutant. The outcome of a test case execution is determined by manually comparing the actual output model with the expected output model. A test case execution fails when the actual and expected models are different, or a runtime exception occurs. A passed test case execution produces an actual output same as the expected output. For a given mutant, if a test case execution fails, we conclude that the mutant is killed, and we move on to the next mutant. If none of the test case executions fail for a given mutant, we conclude that the mutant is alive.

The module contains 1 matched rule and 10 lazy rules. Therefore, the application of M2L and L2M operators resulted in the generation of one lazy rule and 10 matched rules, respectively. The DFE operator was not used because the matched rule *URNDefinition_To_UMLPackage* (see Listing 22) did not contain a filtering expression. The AFE operator also could not be applied on the matched rule because the source object did not contain any scalar attribute that could be used to create a filtering expression. Because the module contains declarative rules, the application of the CEM operator will result in syntactically incorrect mutants. Therefore, the CEM operator was not used for mutant generation. A DAM mutant was created for each of the 37 attribute mappings in the module. Because all the source objects had no unmapped attributes, the AAM mutant was not applicable. Table 19 shows one CST and one CTT mutant created for the *Responsibility_To_OpaqueAction* lazy rule (see Listing 23).

Table 19: CST and CTT mutant corresponding to lazy rule *Responsibility_To_OpaqueAction*

CST mutant	CTT mutant
<pre> lazy rule Responsibility_To_OpaqueAction { from r: UCM!"ucm::map::WaitingPoint" to a: UML!OpaqueAction (name <- r.respDef.name) } </pre>	<pre> lazy rule Responsibility_To_OpaqueAction { from r: UCM!"ucm::map::RespRef" to a: UML!MergeNode (name <- r.respDef.name) } </pre>

Based on the equations 1 and 2, introduced in Section 6.3, 165 CST mutants (i.e., $(16-1)*(1+10) = 165$) and 171 CTT (i.e., $(10-1)*(1+10+8) = 171$) mutants are generated. The number of DRS mutants corresponds to 12 return statements in the original module.

6.5.3 Mutation Analysis Results

The results of the mutation analysis, presented in Table 20, reveal that 177 mutants are killed by the given seven test cases, and 218 mutants remain alive. The test cases are able to kill all M2L and L2M mutants. Since matched rules cannot be invoked, L2M mutants are killed as a result of runtime failures. 12 of the live DAM mutants correspond to rules which are involved in transforming objects of type *Stub* having the *dynamic* attribute set true. Since the ASCC criterion does not consider attributes, these rules are not exercised by the test cases; thus, their corresponding mutants stay live. Similarly, 45 CTT mutants and 8 DRS mutants stay live. All the 150 live CST mutants correspond to lazy rules, and are equivalent mutants; they cannot be killed by any test case.

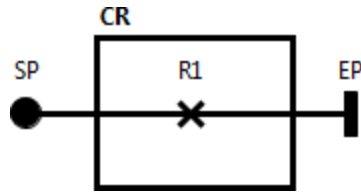
Table 20: Types of mutants created for the UCM to UML 2 AD model transformation

Mutant type	Number of generated mutants	Number of living mutants	Number of killed mutants
M2L	1	0	1
L2M	10	0	10
DAM	37	15	22
CST	165	150	15
CTT	171	45	126
DRS	11	8	3
Total	395	218	177

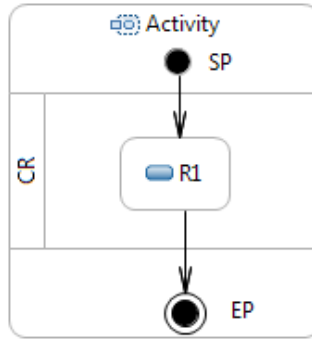
A TS_{eff} score of 72.24% is acquired for the seven test cases. The obtained results show that the proposed mutation operators can effectively determine inadequacies in a test suite.

6.5.4 Test Suite Enhancement

The 68 live non-equivalent mutants (i.e., $218 - 150 = 68$) can be killed by adding new test cases. One DAM mutant will be killed by TC8 (Figure 65), which has a *ComponentRef* object *CR* containing a *RespRef* object *R1*. Similarly, two DAM mutants will be killed by TC9 (Figure 66), which has a *Stub* object *ST* containing a *RespRef* object *R2*.

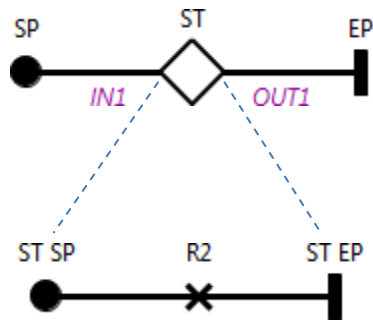


(a) UCM input model



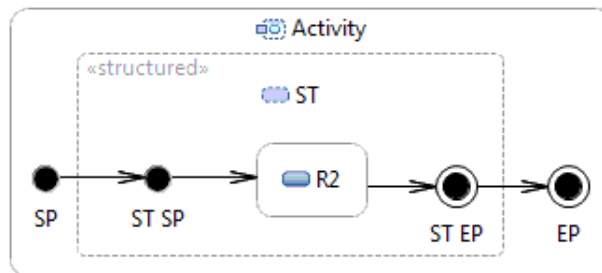
(b) AD expected output model

Figure 65: Input and expected output models of TC8



ST plug-in

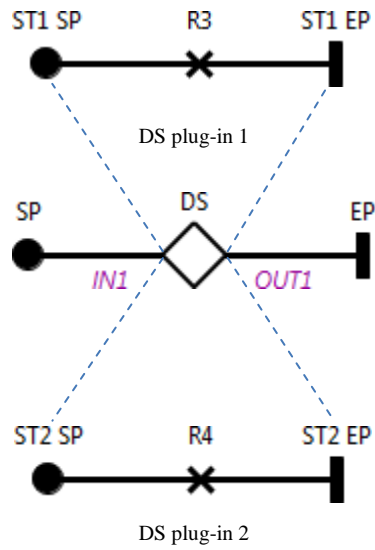
(a) UCM input model



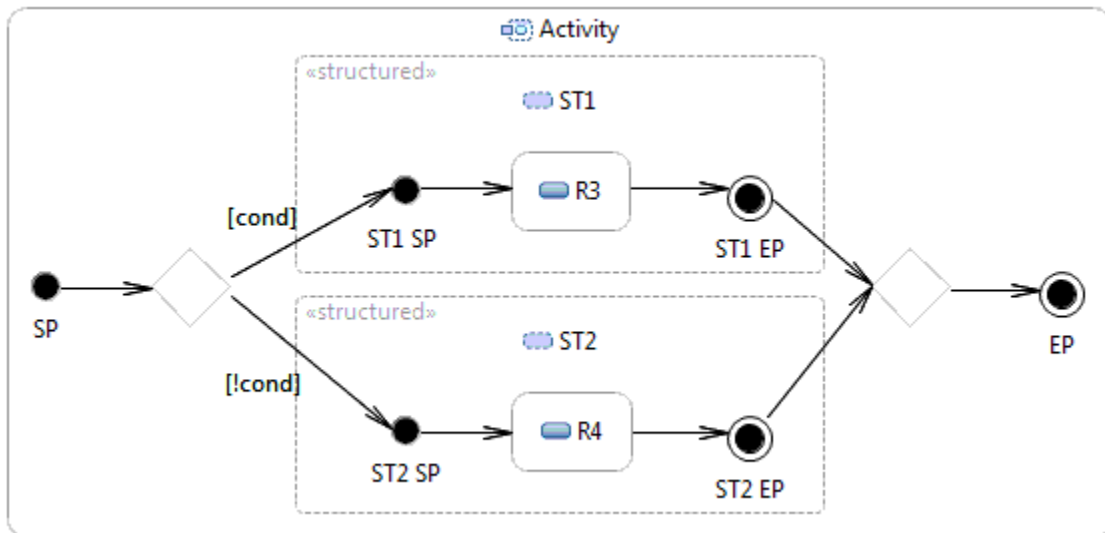
(b) AD expected output model

Figure 66: Input and expected output models of TC9

Adding TC10 (Figure 67), which includes an input model having a *dynamic Stub* object *DS*, and containing *RespRef* objects, *R3* and *R4*, will kill 63 mutants (12 DAM, 45 CTT, and 6 DRS mutants). The *ComponentRef* and *Stub* classes have self associations in the UCM metamodel. Adding, an additional test case (TC11 not shown here) which contains nested *ComponentRef* and nested *Stub* objects will kill the remaining 2 DRS mutants. Adding TC8, TC9, TC10, and TC11 to the test suite gave 100% TS_{eff} .



(a) UCM input model



(b) AD expected output model

Figure 67: Input and expected output models of TC10

6.6 Discussion

Using the proposed mutation operators, we measured the effectiveness of a test suite that corresponds to a UCM to UML 2 AD model transformation. The resulting 72.24% TS_{eff} suggests that the model transformation is not thoroughly verified.

Intuitively, this result was expected because the test suite satisfied a weak coverage criterion, ASCC. 12 DAM, 45 CTT, and 8 DRS live mutants correspond to code fragments, which are not exercised by the test cases. This is an indication that the tester should redesign his test suite or design additional test cases. As stated in Section 6.3.2, the application of CST mutation operator on lazy rules will always produce equivalent mutants. This observation is confirmed in the case study; the 15 killed CST mutants corresponded to a matched rule. Future work should consider defining an operator that is applicable on the actual parameter of a lazy rule.

It must be pointed out that the proposed operators do not consider ATL helpers, which are equivalent to methods in the OO paradigm. The ATL mutation operator set can be enhanced by adding certain method-based Java operators. A complex model transformation's output may make the comparison of expected output and actual output difficult. This problem can be averted by using test oracles, which help in determining the outcome of a test case execution. The test oracles presented in [90] and [135] can be used in conjunction with the approach presented in this paper.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This chapter concludes the thesis by summarizing its contributions, pointing out its limitations, and highlighting future directions of research.

7.1 Thesis Summary

The quality of use case models significantly affects the overall quality of a software product. Defects in a use case model are very likely to propagate to other artifacts, thus resulting in an incorrect implementation of the system. Correction of use case modeling defects at later phases of the development cycle is very expensive. Therefore, early defect correction in use case models is crucial for reducing development costs and improving overall product quality.

In this thesis, we proposed a new technique for improving the quality of use case models, and demonstrated its usage on a real world system. The technique can detect defects in a use case model, and automatically perform improvements. Usage of this approach early in the development cycle will be very beneficial as it prevents propagation of defects to other artifacts. Manual refactoring of complex use case models with hundreds of use cases is susceptible to human error, and is often time consuming. For such use case models, usage of the proposed model transformations will significantly

reduce development time and effort. The application of this technique does not require knowledge of advanced concepts such as metamodeling and OCL. Therefore, inexperienced modelers can easily use this technique to improve the quality of their use case models.

To demonstrate the effectiveness of our approach, a case study that pertains to a bio-diversity system, MAPSTEDI, is presented. The use case models of MASPTEDI contain several quality degrading problems (antipatterns). Four of the presented antipatterns, a2, a4, a5, a8, are detected. This shows that real-world use case models are prone to low quality design and practices. To improve the quality of MAPSTEDI use case models, antipatterns are refactored by executing corresponding model transformations. Antipattern a1 was detected after merging two use case models, *Database Queries* and *Database Integrator*, which contain common entities. The refactoring r1 is finally applied on the merged model to result in a high quality use case model. The refactorings r7, r8, and r14 improve the understandability of use case models, and makes them more analytical. The refactorings r1, r10, and r15 enhance the correctness and consistency of use case models.

This thesis contributes to the MDE software development methodology, which relies on automated transformation of software models. Usage of the proposed UCM to UML 2 AD transformation will enable consistent communication between requirements engineers and designers/developers involved in a software development project. The requirements engineers can model use case scenarios using UCMs. The designers/developers who are not familiar with the UCM notation can use the proposed

transformation to convert UCMs to ADs, which are part of UML, the de-facto standard for documenting design. Moreover, the transformation will aid in minimizing the conceptual gap between the requirements and design.

Furthermore, we presented traceable mappings from UCM to UML 2 SD notation. A systematic approach to derive diagrams from one another also promotes traceability in an OO system. The resulting SDs of the transformation can be refined by the designers, and eventually be converted to source code. Several tools allow automatic code-skeleton generation from SDs. The combined usage of UCMs, the proposed mapping, and code generation tools will allow source code to be easily traced to the scenario definitions.

In a MDE process, model transformations should be thoroughly tested to ensure product quality, and to reduce costs. Mutation testing has been extensively studied in the literature and shown to be more effective than coverage based techniques. To support the usage of mutation testing in MDE, this thesis has defined a set of mutation operators for the ATL model transformation language. The proposed operators are implemented into a tool, called *MuATL*, allowing for automatic generation of ATL mutants. Our approach has been validated using the UCM to UML 2 AD model transformation. The results have shown that the proposed ATL operators can successfully detect inadequacies in an example test suite.

To conclude, this thesis has shown how software developers can embrace the notion of model transformations in the context of FRS by automated refactoring of use case models, and automated derivation of high-level design models from scenario specifications.

7.2 Future Work

A use case model may contain instances of different antipatterns. Future work involves determining an optimal order in which different antipattern instances can be refactored. An optimal order must ensure that the application of a particular refactoring does not result in a new antipattern instance. Use case models of a system may contain common entities (use cases and actors). If one of these models is refactored, it will be transformed into a state which is inconsistent with the other models. Therefore, these models must be merged before performing refactoring. We aim to incorporate an automated model merging technique into our approach. ATL can seamlessly integrate with Java; this will enable us to create a graphical use case refactoring tool. The tool should be able to allow users to define their own antipatterns and corresponding refactorings. In order to determine whether a refactoring is behavior preserving or not, a modeler must consult the corresponding use case descriptions. This is a limitation of our approach which can be addressed by Natural Language Processing techniques. Alternatively, syntax and semantics of use case descriptions can be embedded into an enhanced use case metamodel. This will enable our approach to confirm the presence of antipatterns by automatic analysis of use case descriptions, which conform to the enhanced use case metamodel. Other future work can be directed towards creating model transformations to refactor misuse case models, which are an extension to use case models that allows analysts to specify and communicate the functional security requirements of a system.

The target models (ADs) produced by the transformation are specific to the Eclipse UML 2 tools. Tools, such as Enterprise Architect and Rational Rose allow designers to import/export platform independent models. Our future work involves implementing this mapping to produce platform independent ADs, which can be imported into other platforms. Mapping of UCMs to UML state-chart diagrams is also part of our future work.

The proposed UCM to SD mappings were partially automated due to severe limitations in the UML 2 SD metamodel [141]. SD messages depicted inside fragments are not logically bound to their enclosing fragments. The *CombinedFragment* metaclass, which represents fragments, has no reference to its messages. Fragments rely on their position on the modeling tool's design surface to enclose their messages. In future work, a heavy weight extension [131] of the UML 2 metamodel can be performed to remedy this limitation.

Mutation testing can be more efficiently performed when supported by automated tools. As a future work, we are planning to develop further our prototype tool, *MuATL*, to include a test case execution engine and a test oracle. In addition, we aim at conducting an empirical study to better assess the usefulness and the effectiveness of the proposed ATL operators. Furthermore, we will investigate the addition of mutation operators of traditional programming languages that are relevant to ATL. The idea of mutation testing will also be explored for other model transformation languages, such as QVT, Tefkat, and Epsilon.

References

- [1] M. Abbes, *et al.*, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, 2011, pp. 181-190.
- [2] T. Abdelaziz, *et al.*, "Visualizing a Multiagent-Based Medical Diagnosis System Using a Methodology Based on Use Case Maps," *Multiagent System Technologies*, pp. 545-559, 2004.
- [3] C. B. Achour, *et al.*, "Guiding use case authoring: results of an empirical study," in *Requirements Engineering, 1999. Proceedings. IEEE International Symposium on*, 1999, pp. 36-43.
- [4] S. Adolph, *et al.*, *Patterns for effective use cases*: Addison-Wesley Professional, 2002.
- [5] M. Akiyama, *et al.*, "Supporting design model refactoring for improving class responsibility assignment," *Model Driven Engineering Languages and Systems*, pp. 455-469, 2011.
- [6] D. Amyot, *et al.*, "UCM-driven testing of web applications," *SDL 2005: Model Driven*, pp. 1213-1229, 2005.
- [7] D. Amyot, *et al.*, "Generating scenarios from use case map specifications," in *Third International Conference on Quality Software (QSIC'03)*, 2003, pp. 108-115.
- [8] D. Amyot, "Bridging the gap between requirements and design with use case maps," [online] Available: <http://people.scs.carleton.ca/~jeanpier/304/Amyot.PDF>, 2001 [Feb. 10, 2013]
- [9] D. Amyot and L. Logrippo, "Use case maps and lotos for the prototyping and validation of a mobile group call system," *Computer Communications*, vol. 23, pp. 1135-1157, 2000.
- [10] D. Amyot and G. Mussbacher, "On the Extension of UML with Use Case Maps Concepts," in *«UML» 2000 — The Unified Modeling Language*. vol. 1939, A. Evans, *et al.*, Eds., ed: Springer Berlin Heidelberg, 2000, pp. 16-31.

- [11] B. Anda, and D. I. K. Sjøberg, "Towards an inspection technique for use case models," in *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, 2002, pp. 127-134.
- [12] B. Anda, *et al.*, "Quality and understandability of use case models," *ECOOP 2001—Object-Oriented Programming*, pp. 402-428, 2001.
- [13] E. Anderson, *et al.*, "Use case and business rules: styles of documenting business rules in use cases," in *Conference on Object Oriented Programming Systems Languages and Applications: Addendum to the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications(Addendum)*, 1997, pp. 85-87.
- [14] R. Andrade, "Applying Use Case Maps and Formal Methods to the Development of Wireless Mobile ATM Networks," in *Proc. of the Fifth NASA Langley Formal Methods Workshop*, 2000.
- [15] V. Aranega, *et al.*, "Using trace to situate errors in model transformations," *Software and Data Technologies*, pp. 137-149, 2011.
- [16] D. Arcelli, *et al.*, "Antipattern-based model refactoring for software performance improvement," in *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*, 2012, pp. 33-42.
- [17] F. Armour and G. Miller, *Advanced use case modeling: software systems*: Addison-Wesley Professional, 2000.
- [18] M. El-Attar and J. Miller, "Constructing high quality use case models: a systematic review of current practices," *Requirements Engineering*, vol. 17, pp. 187-201, 2012.
- [19] M. El-Attar and J. Miller, "Improving the quality of use case models using antipatterns," *Software and Systems Modeling*, vol. 9, pp. 141-160, 2010.
- [20] M. El-Attar, "Improving the Quality of Use Case Models and their Utilization in Software Development," Ph.D. thesis, Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Alberta, 2009.
- [21] M. El-Attar and J. Miller, "Producing robust use case diagrams via reverse engineering of use case descriptions," *Software and Systems Modeling*, vol. 7, pp. 67-83, 2008.

- [22] M. El-Attar and J. Miller, "Matching Antipatterns to Improve the Quality of Use Case Models," in *Requirements Engineering, 14th IEEE International Conference*, 2006, pp. 99-108.
- [23] D. Ballis, *et al.*, "A rule-based method to match Software Patterns against UML Models," *Electronic Notes in Theoretical Computer Science*, vol. 219, pp. 51-66, 2008
- [24] D. Ballis, *et al.*, "A minimalist visual notation for design patterns and antipatterns," in *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, 2008, pp. 51-56.
- [25] B. Baudry, *et al.*, "Barriers to systematic model transformation testing," *Communications of the ACM*, vol. 53, pp. 139-143, 2010
- [26] B. Baudry, *et al.*, "Model transformation testing challenges," in *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing*, 2006.
- [27] E. Bauer and J. Küster, "Combining specification-based and code-based coverage for model transformation chains," *Theory and Practice of Model Transformations*, pp. 78-92, 2011.
- [28] E. Bauer, *et al.*, "Test suite quality for model transformation chains," *Objects, Models, Components, Patterns*, pp. 3-19, 2011.
- [29] B. Berenbach, "The evaluation of large, complex UML analysis and design models," in *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 2004, pp. 232-241.
- [30] R. Biddle, *et al.*, "Essential use cases and responsibility in object-oriented development," *Australian Computer Science Communications*, vol. 24, pp. 7-16, 2002.
- [31] E. A. Billard, "Operating system scenarios as Use Case Maps," *ACM SIGSOFT Software Engineering Notes*, vol. 29, pp. 266-277, 2004.
- [32] K. Bittner and I. Spence, *Use case modeling*: Addison-Wesley Professional, 2003.
- [33] F. Bordeleau and D. Cameron, "On the relationship between use case maps and message sequence charts," in *2nd Workshop of the SDL Forum Society on SDL and MSC (SAM2000)*, 2000.

- [34] F. Bordeleau and R. J. A. Buhr, "UCM-ROOM modelling: from use case maps to communicating state machines," in *Engineering of Computer-Based Systems, 1997. Proceedings., International Conference and Workshop on*, 1997, pp. 169-178.
- [35] E. Brottier, *et al.*, "Metamodel-based test generation for model transformations: an algorithm and a tool," in *Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on*, 2006, pp. 85-94.
- [36] W. J. Brown, *AntiPatterns: refactoring software, architectures, and projects in crisis*: Wiley, 1998.
- [37] F. Budinsky, *Eclipse modeling framework: a developer's guide*: Addison-Wesley Professional, 2004.
- [38] R. J. A. Buhr, "Use case maps as architectural entities for complex systems," *Software Engineering, IEEE Transactions on*, vol. 24, pp. 1131-1155, 1998.
- [39] R. J. A. Buhr and R. S. O. Casselman, *Use case maps for object-oriented systems*: Prentice Hall, 1996.
- [40] G. Butler and L. Xu, "Cascaded refactoring for framework," *ACM SIGSOFT Software Engineering Notes*, vol. 26, pp. 51-57, 2001.
- [41] J. Cabot, *et al.*, "Verification and validation of declarative model-to-model transformations through invariants," *Journal of Systems and Software*, vol. 83, pp. 283-302, 2010.
- [42] A. Cailliau, "Automating Model Transformation and Refactoring for Goal-Oriented Models", M.S thesis, Department of Computer Engineering, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2010.
- [43] P. Chandrasekaran, "How use case modeling policies have affected the success of various projects (or how to improve use case modeling)," in *Addendum to the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (Addendum)*, 1997, pp. 6-9.
- [44] A. Ciancone, *et al.*, "Mantra: Towards model transformation testing," in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, 2010, pp. 97-105.
- [45] A. Cockburn, *Writing effective use cases* vol. 1: Addison-Wesley Boston, 2001.

- [46] L. L. Constantine, "Essential modeling: Use cases for user interfaces," *interactions*, vol. 2, pp. 34-46, 1995.
- [47] V. Cortellessa, et al., "Digging into UML models to remove performance antipatterns," in *Proc. of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems (Quovadis' 10)*, Cape Town, South Africa, 2010.
- [48] V. Cortellessa, et al., "Performance antipatterns as logical predicates," in *Engineering of Complex Computer Systems (ICECCS), 2010 15th IEEE International Conference on*, 2010, pp. 146-156.
- [49] K. Cox and K. Phalp, "Replicating the CREWS Use Case Authoring Guidelines Experiment," *Empirical Software Engineering*, vol. 5, pp. 245-267, 2000.
- [50] K. Czarnecki and S. Helsen, "Classification of model transformation approaches," in *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, 2003, pp. 1-17.
- [51] I. Deligiannis, *et al.*, "A controlled experiment investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 72, pp. 129-143, 2004.
- [52] I. Deligiannis, *et al.*, "An empirical investigation of an object-oriented design heuristic for maintainability," *Journal of Systems and Software*, vol. 65, pp. 127-139, 2003.
- [53] B. Demuth, *et al.*, "Experiments with XMI based transformations of software models," in *Workshop on Transformations in UML*, 2001.
- [54] K. Dhambri, et al., "Visual detection of design anomalies," in *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, 2008, pp. 279-283.
- [55] Ł. Dobrzański and L. Kuźniarz, "An approach to refactoring of executable UML models," in *Proceedings of the 2006 ACM symposium on Applied computing*, 2006, pp. 1273-1279.
- [56] B. Du Bois, *et al.*, "Does God Class Decomposition Affect Comprehensibility?," in *IASTED International Conference on Software Engineering*, 2006.
- [57] H. Einarsson and H. Neukirchen, "An approach and tool for synchronous refactoring of UML diagrams and models using model-to-model transformations," in *Proceedings of the Fifth Workshop on Refactoring Tools*, 2012, pp. 16-23.

- [58] H. Einarsson, "Refactoring UML Diagrams and Models with Model-to-Model Transformations," M.S thesis, Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland, Reykjavík, Iceland, 2011.
- [59] M. Elammari and W. Lalonde, "An agent-oriented methodology: High-level and intermediate models," in *Proc. of the 1st Int. Workshop. on Agent-Oriented Information Systems*, 1999, pp. 1-16.
- [60] J. Ellsberger, et al., *SDL: formal object-oriented language for communicating systems*: Prentice Hall, 1997.
- [61] T. v. Enckevoort, "Refactoring UML models: using OpenArchitectureWare to measure UML model quality and perform pattern matching on UML models with OCL queries," in *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, 2009, pp. 635-646.
- [62] F. Fabbrini, *et al.*, "The linguistic approach to the natural language requirements quality: benefit of the use of an automatic tool," in *Software Engineering Workshop, 2001. Proceedings. 26th Annual NASA Goddard*, 2001, pp. 97-105.
- [63] A. Fantechi, *et al.*, "Application of linguistic techniques for Use Case analysis," in *Requirements Engineering, 2002. Proceedings. IEEE Joint International Conference on*, 2002, pp. 157-164.
- [64] M. Favre, "Towards a basic theory to model driven engineering," in *3rd Workshop in Software Model Engineering, WiSME*, 2004.
- [65] C. Fiorentini, *et al.*, "A constructive approach to testing model transformations," *Theory and Practice of Model Transformations*, pp. 77-92, 2010.
- [66] D. G. Firesmith, "Use case modeling guidelines," in *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30. Proceedings*, 1999, pp. 184-193.
- [67] F. Fleurey, *et al.*, "Qualifying input test data for model transformations," *Software and Systems Modeling*, vol. 8, pp. 185-203, 2009.
- [68] F. Fleurey, *et al.*, "Validation in model-driven engineering: testing model transformations," in *Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on*, 2004, pp. 29-40.

- [69] A. Folli and T. Mens, "Refactoring of UML models using AGG," *Electronic Communications of the EASST*, vol. 8, 2008.
- [70] R. Fourati, *et al.*, "A Metric-Based Approach for Anti-pattern Detection in UML Designs," *Computer and Information Science 2011*, pp. 17-33, 2011.
- [71] M. Fowler, *et al.*, *Refactoring: Improving the Design of Existing Code*: Pearson Education, 2012.
- [72] R. France and J. M. Bieman, "Multi-view software evolution: a UML-based framework for evolving object-oriented software," in *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, 2001, pp. 386-395.
- [73] P. Giner and V. Pelechano, "Test-driven development of model transformations," *Model Driven Engineering Languages and Systems*, pp. 748-752, 2009.
- [74] H. Gomaa, *Designing concurrent, distributed, and real-time applications with UML*: Addison-Wesley, 2000.
- [75] H. Gomaa, "Use cases for distributed real-time software architectures," in *Parallel and Distributed Real-Time Systems, 1997. Proceedings of the Joint Workshop on*, 1997, pp. 34-42.
- [76] C. González and J. Cabot, "ATLTest: A White-Box Test Generation Approach for ATL Transformations," *Model Driven Engineering Languages and Systems*, pp. 449-464, 2012.
- [77] E. Guerra, "Specification-driven test generation for model transformations," *Theory and Practice of Model Transformations*, pp. 40-55, 2012.
- [78] R. J. Harwood, "Use case formats: Requirements, analysis, and design," *JOOP*, vol. 9, pp. 54-57, 1997.
- [79] Y. He, *et al.*, "Synthesizing SDL from use case maps: an experiment," *SDL 2003: System Design*, pp. 159-159, 2003.
- [80] A. Issa, "Utilising refactoring to restructure use-case models," in *Proc. of the World Congress on Engineering*, 2007, pp. 523-527.
- [81] I. Ivkovic and K. Kontogiannis, "A framework for software architecture refactoring using model transformations and semantic annotations," in *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, 2006, pp. 10 pp.-144.

- [82] ITU-T, "Recommendation Z.151, User Requirements Notation (URN)" [online] Available: <http://www.itu.int/rec/T-REC-Z.151/en>, 2010 [Feb. 10, 2013]
- [83] A. Jaaksi, "Our cases with use cases," *JOOP*, vol. 10, pp. 58-65, 1998.
- [84] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *Software Engineering, IEEE Transactions on*, vol. 37, pp. 649-678, 2011.
- [85] F. Jouault, *et al.*, "ATL: A model transformation tool," *Science of Computer Programming*, vol. 72, pp. 31-39, 2008.
- [86] F. Jouault and I. Kurtev, "Transforming models with ATL," in *Satellite Events at the MoDELS 2005 Conference*, 2006, pp. 128-138.
- [87] "jUCMNav" [online] Available: <http://www.ohloh.net/p/11712>, 2013 [Feb. 10, 2013]
- [88] J. Kealey and D. Amyot, "Towards the automated conversion of natural-language use cases to graphical use case maps," in *Electrical and Computer Engineering, 2006. CCECE'06. Canadian Conference on*, 2006, pp. 2377-2380.
- [89] S. Kent, "Model driven engineering," in *Integrated Formal Methods*, 2002, pp. 286-298.
- [90] M. Kessentini, *et al.*, "Example-based model-transformation testing," *Automated Software Engineering*, vol. 18, pp. 199-224, 2011.
- [91] M. Kessentini, *et al.*, "Deviance from perfection is a better criterion than closeness to evil when identifying risky code," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, 2010, pp. 113-122.
- [92] Y. A. Khan, "Antipattern based Use Case Refactoring" [online] Available: <https://sourceforge.net/projects/apucrefactoring/>, 2012 [Feb. 10, 2013]
- [93] Y. A. Khan and M. El-Attar, "Automated transformation of use case maps to uml activity diagrams," in *ICSOFT*, S. Hammoudi, M. van Sinderen, and J. Cordeiro, Eds. SciTePress, 2012, pp. 184-189.
- [94] Y. A. Khan, "Transforming UCMs to ADs - the ATL Code" [online] Available: <https://sourceforge.net/projects/ucmtoumlad/>, 2012 [Feb. 10, 2013]
- [95] Y. Kim and K.-G. Doh, "The service modeling process based on use case refactoring," in *Business information systems*, 2007, pp. 108-120.

- [96] A. G. Kleppe, et al., *MDA Explained, the Model Driven Architecture: Practice and Promise*: Addison-Wesley, 2003.
- [97] D. S. Kolovos, et al., "Update transformations in the small with the epsilon wizard language," *Journal of Object Technology (JOT)*, 2003.
- [98] P. Kroll and P. Kruchten, *The rational unified process made easy: a practitioner's guide to the RUP*: Addison-Wesley, 2003.
- [99] P. Kruchten, "Modeling component systems with the unified modeling language," in *International Workshop on Component-Based Software Engineering*, 1998.
- [100] F. Khomh, et al., "An exploratory study of the impact of antipatterns on class change-and fault-proneness," *Empirical Software Engineering*, vol. 17, pp. 243-275, 2012.
- [101] F. Khomh, et al., "BDTEX: A GQM-based Bayesian approach for the detection of antipatterns," *Journal of Systems and Software*, vol. 84, pp. 559-572, 2011.
- [102] J. Kovse and T. Härder, "Generic XMI-based UML model transformations," *Object-Oriented Information Systems*, pp. 183-190, 2002.
- [103] D. Kulak and E. Guiney, *Use cases: requirements in context*: Addison-Wesley Professional, 2004.
- [104] J. Küster, et al., "Incremental development of model transformation chains using automated testing," *Model Driven Engineering Languages and Systems*, pp. 733-747, 2009.
- [105] J. Küster and M. Abd-El-Razik, "Validation of model transformations—first experiences using a white box approach," *Models in Software Engineering*, pp. 193-204, 2007.
- [106] M. Lamari, "Towards an automated test generation for the verification of model transformations," in *Proceedings of the 2007 ACM symposium on Applied computing*, 2007, pp. 998-1005.
- [107] G. Langelier, et al., "Visualization-based analysis of quality for large-scale software systems," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, 2005, pp. 214-223.
- [108] M. Lawley, "Tefkat: The EMF Transformation Engine" [online] Available: <http://tefkat.sourceforge.net/> [Feb. 10, 2013]

- [109] K. Li, *et al.*, "A Generic Technique for Domain-Specific Visual Language Model Refactoring to Patterns," *Electronic Communications of the EASST*, vol. 31, 2011.
- [110] W. Li and R. Shatnawi, "An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution," *Journal of Systems and Software*, vol. 80, pp. 1120-1128, 2007.
- [111] S. Lilly, "Use case pitfalls: top 10 problems from real projects using use cases," in *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30. Proceedings*, 1999, pp. 174-183.
- [112] Y. Lin, *et al.*, "A testing framework for model transformations," *Model-Driven Software Development-Research and Practice in Software Engineering*, pp. 219-236, 2005.
- [113] H. Liu, *et al.*, "Detecting overlapping use cases," *Software, IET*, vol. 1, pp. 29-36, 2007.
- [114] Y.-S. Ma, *et al.*, "MuJava: an automated class mutation system: Research Articles," *Softw. Test. Verif. Reliab.*, vol. 15, pp. 97-133, 2005.
- [115] A. Maiga, *et al.*, "SMURF: A SVM-based Incremental Anti-pattern Detection Approach," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, 2012, pp. 466-475.
- [116] "MAPSTEDI" [online] Available: <http://mapstedi.sourceforge.net/> [Feb. 10, 2013]
- [117] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, 2004, pp. 350-359.
- [118] S. Markovic, "Model refactoring using transformations," Ph.D. thesis, Faculty of Computer and Communications, Federal Polytechnic School of Lausanne, Lausanne, Switzerland, 2008.
- [119] S. Markovic and T. Baar, "Refactoring OCL annotated UML class diagrams," *Model Driven Engineering Languages and Systems*, pp. 280-294, 2005.
- [120] T. Massoni, *et al.*, "Formal model-driven program refactoring," *Fundamental Approaches to Software Engineering*, pp. 362-376, 2008.

- [121] L. Mattingly and H. Rao, "Writing effective use cases and introducing collaboration cases," *Journal of Object Oriented Programming*, vol. 11, pp. 77-84, 1998.
- [122] H. Martínez, "Synthesizing state-machine behaviour from UML collaborations and Use Case Maps," *SDL 2005: Model Driven*, pp. 1192-1195, 2005.
- [123] J. R. McCoy, "Requirements use case tool (RUT)," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2003, pp. 104-105.
- [124] M. J. McGill and B. H. C. Cheng, "Test-driven development of a model transformation with jemtte," Technical Report, Software Engineering and Network Systems Laboratory, Department of Computer Science and Engineering, Michigan State University, 2007.
- [125] J. A. McQuillan and J. F. Power, "White-box coverage criteria for model transformations," *Model Transformation with ATL*, p. 63, 2009.
- [126] N. Medvidovic, *et al.*, "Modeling software architectures in the Unified Modeling Language," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 11, pp. 2-57, 2002.
- [127] T. Mens, *et al.*, "Challenges in model refactoring," in *Proc. 1st Workshop on Refactoring Tools, University of Berlin*, 2007.
- [128] T. Mens, *et al.*, "Applying a model transformation taxonomy to graph transformation technology," *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 143-159, 2006.
- [129] T. Mens, *et al.*, "Detecting structural refactoring conflicts using critical pair analysis," *Electronic Notes in Theoretical Computer Science*, vol. 127, pp. 113-128, 2005.
- [130] Miga, *et al.*, "Deriving message sequence charts from use case maps scenario specifications," in *SDL 2001: Meeting UML*, 2001, pp. 268-287.
- [131] M. Misbhauddin and M. Alshayeb, "Extending the UML Metamodel for Sequence Diagram to Enhance Model Traceability," in *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*, 2010, pp. 129-134.
- [132] N. Moha, *et al.*, "DECOR: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, pp. 20-36, 2010.

- [133] N. Moha, *et al.*, "Evaluation of Kermeta for solving graph-based problems," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 12, pp. 273-285, 2010.
- [134] J.-M. Mottu, *et al.*, "Static Analysis of Model Transformations for Effective Test Generation," in *ISSRE-23rd IEEE International Symposium on Software Reliability Engineering*, 2012.
- [135] J. M. Mottu, *et al.*, "Model transformation testing: oracle issue," in *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on, 2008*, pp. 105-112.
- [136] J.-M. Mottu, *et al.*, "Mutation analysis testing for model transformations," in *Model Driven Architecture–Foundations and Applications*, 2006, pp. 376-390.
- [137] O. Muliawan and D. Janssens, "Model refactoring using MoTMoT," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 12, pp. 201-209, 2010.
- [138] M. J. Munro, "Product Metrics for Automatic Identification of Bad Smells Design Problems in Java Source-Code" in *Software Metrics, 2005. 11th IEEE International Symposium*, 2005, pp. 15-15.
- [139] G. Mussbacher and D. Amyot, "Assessing the applicability of use case maps for business process and workflow description," in *e-Technologies, 2008 International MCETECH Conference on*, 2008, pp. 219-222
- [140] Object Management Group, "Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification," [online] Available: <http://www.omg.org/spec/QVT/1.1/PDF/>, 2011 [Feb. 10, 2013]
- [141] Object Management Group, "OMG Unified Modeling Language Superstructure Specification," [online] Available: <http://www.omg.org/spec/UML/2.2/Superstructure/PDF/>, 2009 [Feb. 10, 2013]
- [142] Object Management Group, "Meta Object Facility (MOF) Core Specification," [online] Available: <http://www.omg.org/spec/MOF/2.0/PDF/>, 2006 [Feb. 10, 2013]
- [143] S. Olbrich, *et al.*, "The evolution and impact of code smells: A case study of two open source systems," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 390-400.

- [144] R. Oliveto, *et al.*, "Numerical signatures of antipatterns: An approach based on b-splines," in *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, 2010, pp. 248-251.
- [145] W. F. Opdyke, *Refactoring Object-Oriented Frameworks*, Ph.D. thesis, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1992.
- [146] G. Övergaard and K. Palmkvist, *Use cases: patterns and blueprints*: Addison-Wesley, 2005.
- [147] D. Petriu, *et al.*, "Traceability and evaluation in scenario analysis by use case maps," *Scenarios: Models, Transformations and Tools*, pp. 574-575, 2005.
- [148] D. Petriu and M. Woodside, "Software performance models from system scenarios in use case maps," *Computer Performance Evaluation: Modelling Techniques and Tools*, pp. 1-8, 2002.
- [149] F. Pettersson, *et al.*, "Automotive use case standard for embedded systems," *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 1-6, 2005.
- [150] A. Pleuss, *et al.*, "Model-driven support for product line evolution on feature level," *Journal of Systems and Software*, 2011.
- [151] K. Pohl, *Requirements Engineering: Fundamentals, Principles, and Techniques*: Springer, 2010.
- [152] I. Porres, "Rule-based update transformations and their application to model refactorings," *Software and Systems Modeling*, vol. 4, pp. 368-385, 2005.
- [153] I. Porres, *et al.*, "Model refactorings as rule-based update transformations," «UML» 2003-*The Unified Modeling Language. Modeling Languages and Applications*, pp. 159-174, 2003.
- [154] R. S. Pressman, *Software engineering: a practitioner's approach*: McGraw-Hill Higher Education, 2010.
- [155] R. Ramos, *et al.*, "Improving the Quality of Requirements with Refactoring," *VI Simpósio Brasileiro de Qualidade de Software—SBQS2007, Porto de Galinhas, Recife, Pernambuco, Brasil*, 2007.
- [156] D. Romano, *et al.*, "Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, 2012, pp. 437-446.

- [157] D. Rosenberg and K. Scott, *Use case driven object modeling with UML: a practical approach*: Addison-Wesley, 1999.
- [158] K. Rui and G. Butler, "Refactoring use case models: the metamodel," in *Proceedings of the 26th Australasian computer science conference-Volume 16*, 2003, pp. 301-308.
- [159] J. Rumbaugh, et al., *The unified modeling language reference manual*: Addison-Wesley, 2005.
- [160] O. Ryndia and P. Kritzing, "Improving Requirements Specification Verification of Use Case Models with Susan," Technical Report CS04-06-00, Department of Computer Science, University of Cape Town, 2004.
- [161] I. Sales and R. Probert, "From high-level behaviour to high-level design: Use case maps to specification and description language," *SBRC'2000, 18 Simpósio Brasileiro de Redes de Computadores*, 2000.
- [162] D. C. Schmidt, "Model-driven engineering," *Computer-IEEE Computer Society*, vol. 39, p. 25, 2006.
- [163] G. Schneider and J. P. Winters, *Applying use cases: a practical guide*: Addison-Wesley, 2001.
- [164] S. Segura, *et al.*, "Automated merging of feature models using graph transformations," *Generative and Transformational Techniques in Software Engineering II*, pp. 489-505, 2008
- [165] B. Selic, *Real-time object-oriented modeling*: Wiley & Sons, 1994.
- [166] G. M. Selim, *et al.*, "Model transformation testing: the state of the art," in *Proceedings of the First Workshop on the Analysis of Model Transformations, Innsbruck, Austria*, 2012, pp. 21-26.
- [167] S. Sen, *et al.*, "Using Models of Partial Knowledge to Test Model Transformations," *Theory and Practice of Model Transformations*, pp. 24-39, 2012.
- [168] S. Sen, *et al.*, "Automatic model generation strategies for model transformation testing," *Theory and Practice of Model Transformations*, pp. 148-164, 2009.
- [169] S. Sendall and W. Kozaczynski, "Model transformation: The heart and soul of model-driven software development," *Software, IEEE*, vol. 20, pp. 42-45, 2003.

- [170] D. Settas, *et al.*, "Enhancing ontology-based antipattern detection using Bayesian networks," *Expert Systems with Applications*, 2012.
- [171] F. Simon, *et al.*, "Metrics based refactoring," in *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, 2001, pp. 30-38.
- [172] T. Stahl *et al.*, *Model-Driven Software Development: Technology, Engineering, Management*: John Wiley & Sons, 2006.
- [173] M. Stephan and J. R. Cordy, "Application of model comparison techniques to model transformation testing," *MODELSWARD*. *to appear*, 2013.
- [174] A. Stoianov and I. Sora, "Detecting patterns and antipatterns in software using Prolog rules," in *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, 2010, pp. 253-258.
- [175] The Eclipse Foundation, "ATL – A Model Transformation Technology" [online] Available: <http://www.eclipse.org/atl/>, 2013 [Feb. 10, 2013]
- [176] The Eclipse Foundation, "Eclipse Indigo" [online] Available: <http://www.eclipse.org/indigo/>, 2013 [Feb. 10, 2013]
- [177] The Eclipse Foundation, "MDT-UML2Tools" [online] Available: <http://wiki.eclipse.org/MDT-UML2Tools>, 2013 [Feb. 10, 2013]
- [178] The Eclipse Foundation, "Epsilon" [online] Available: <http://www.eclipse.org/epsilon/>, 2012 [Feb. 10, 2013]
- [179] G. Travassos, *et al.*, "Detecting defects in object-oriented designs: using reading techniques to increase software quality," *ACM Sigplan Notices*, vol. 34, pp. 47-56, 1999.
- [180] C. Trubiani and A. Koziolk, "Detection and solution of software performance antipatterns in palladio architectural models," in *ACM SIGSOFT Software Engineering Notes*, 2011, pp. 19-30.
- [181] UModel - UML tool for software modeling and application development [online] Available: <http://www.altova.com/umodel.html>, 2013 [Feb. 10, 2013]
- [182] E. Van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, 2002, pp. 97-106.

- [183] D. Wagelaar, *et al.*, "Module superimposition: a composition technique for rule-based model transformation languages," *Software and Systems Modeling*, vol. 9, pp. 285-309, 2010.
- [184] J. Wang, *et al.*, "Verifying metamodel coverage of model transformations," in *Software Engineering Conference, 2006. Australian*, 2006, p. 10 pp.
- [185] T. Weilkiens and B. Oestereich, *UML 2 certification guide: fundamental and intermediate exams*: Morgan Kaufmann, 2007.
- [186] R. Wieman, *Anti-Pattern Scanner: An Approach to Detect Anti-Patterns and Design Violations*: LAP Lambert Academic Publishing, 2011.
- [187] R. Wirfs-Brock, "Designing scenarios: Making the case for a use case framework," *The Smalltalk Report*, vol. 3, 1993.
- [188] L. Xu and G. Butler, "Cascaded refactoring for framework development and evolution," in *Software Engineering Conference, 2006. Australian*, 2006, p. 10 pp.
- [189] L. Xu, *et al.*, "Use case refactoring: a tool and a case study," in *Software Engineering Conference, 2004. 11th Asia-Pacific*, 2004, pp. 484-491.
- [190] W. Xue-Bin, *et al.*, "Research and implementation of design pattern-oriented model transformation," in *Computing in the Global Information Technology, 2007. ICCGI 2007. International Multi-Conference on*, 2007, pp. 24-24.
- [191] W. Yu, *et al.*, "Refactoring use case models on episodes," in *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, 2004, pp. 328-335.
- [192] Y. X. Zeng, "Transforming Use Case Maps to the Core Scenario Model Representation," M.S thesis, Ottawa-Carleton Institute for Computer Science, University of Ottawa, Ottawa, Ontario, 2005.
- [193] J. Zhang, *et al.*, "Generic and domain-specific model refactoring using a model transformation engine," *Model-driven Software Development*, pp. 199-218, 2005.

Appendix B – UML 2 AD Metamodel

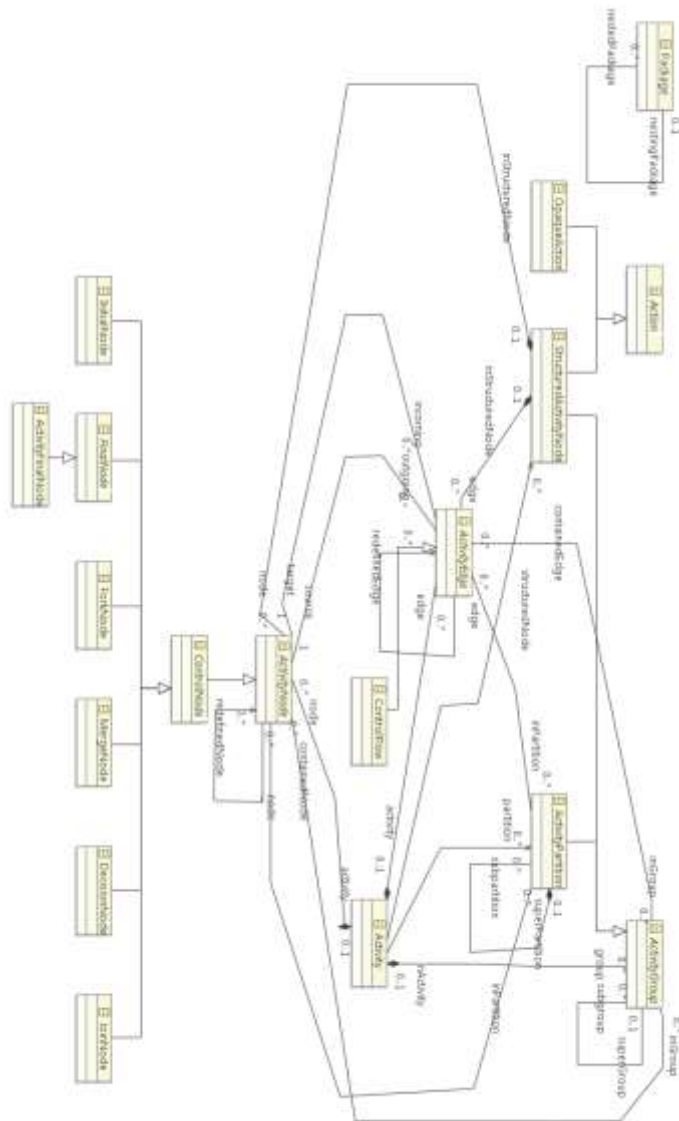


Figure 69: UML 2 AD metamodel

Vitae

Name	Yasser Ali Khan
Nationality	Indian
Date of Birth	3/10/1987
Email	ybakhan@gmail.com
Current Address	802-308, KFUPM, 31261, Saudi Arabia
Permanent Address	H.No. 16-4-315, Chanchalguda, Hyderabad, 500024, Andhra Pradesh, India
Academic Background	Bachelor of Science in Computer Science at King Fahd University of Petroleum & Minerals, Dhahran, Kingdom of Saudi Arabia
Mobile	+966 562102680