TOOL SUPPORT FOR MODEL COMPLIANCE VERIFICATION

A QVT Based Approach

by

Tobin McClean

B.C.S.

A thesis submitted to the
Department of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Masters of Computer Science

Ottawa-Carleton Institute for Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario

2005

© Copyright 2005, Tobin McClean

Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file   Votre référence

ISBN: 0-494-00763-X

Our file   Notre référence

ISBN: 0-494-00763-X

NOTICE:

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

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

AVIS:

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

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

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

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

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

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

Canada

DEDICATION

This work is dedicated to my aunt, Nancy Norton, who left us way to early.

ABSTRACT

The advances in Model Driven Development (MDD) most notably the OMG's Model Driven Architecture (MDA) provide a framework that helps bridge the gap between a specification model and a design model. Ideally a specification model would be transformed into a design model, which can then be augmented and transformed into the executing system, always retaining compliance with the specification. This ideal situation is seldom reality. That is, the design model is created by transforming the specification model and then modified. These modifications can affect the design models' compliance with the specification model. Therefore, techniques and tools are needed so that a software engineer is able to verify that his/her design model complies with the properties and capabilities defined in the specification model. This dissertation addresses the feasibility of providing tool support for model compliance verification between a specification model and a design model using what will become the defacto model-to-model transformation language: OMG's Query, Views, and Transformations (QVT) standard.

ACKNOWLEDGEMENTS

I am most grateful for the guidance and encouragement provided by my supervisors: Dr. Jean-Pierre Corriveau and Dr. Francis Bordeleau. Dr. Corriveau has spent countless hours guiding my research, discussing software engineering research, and reviewing this dissertation. This time and effort is greatly appreciated. Dr. Bordeleau has provided the opportunity to meet and learn from some of the leading software engineering researchers, and also important feedback throughout my research. Without the help of both Dr. Corriveau and Dr. Bordeleau the completion of this dissertation would not have been possible.

A special thanks to my fellow software engineering researchers: David Arnold and Juan Pablo Zamora Zapata. Dave has provided a great deal of help in the implementation of the prototype. And without Juan Pablo's research in developing a framework for model compliance verification this dissertation would not have been possible.

I am indebted to the National Sciences and Engineering Research Council of Canada (NSERC), Nortel Networks, and Carleton University (School of Computer Science and Faculty of Graduate Studies) for their substantial monetary support through out my graduate work. This support has definitely made it easier to perform and focus on my research.

To my mother, father and sister who have taught me that hard work and perseverance pays off in the end. They have shown patience and understanding throughout the process of completing the research and writing this dissertation. Without their support, encouragement and dedication I would not be where I am today. Thank you mom, dad and Katie.

Finally and most importantly, Lindsay, who has been both patient and positive throughout my graduate work. She has made many sacrifices allowing me to complete and achieve my goals. For this I am most appreciative.

CONTENTS

CHAPTER 1.....	1
INTRODUCTION.....	1
1.1 CONTEXT AND MOTIVATION.....	1
1.1.1 MODELS	1
1.1.2 STANDARD SPECIFICATION – PRODUCT DEVELOPMENT COMPLIANCE.....	2
1.1.3 TOOL SUPPORT.....	3
1.2 PROBLEM STATEMENT.....	4
1.3 PROPOSED SOLUTION	5
1.4 VALIDATION OF PROPOSED SOLUTION.....	6
1.5 CONTRIBUTIONS	6
1.6 OUTLINE OF DISSERTATION.....	7
CHAPTER 2	9
BACKGROUND.....	9
2.1 SPECIFICATION VS. DESIGN	9
2.1.1 SPECIFICATION.....	9
2.1.2 DESIGN.....	10
2.1.3 SPECIFICATION VS. DESIGN.....	10
2.2 MODEL COMPLIANCE VERIFICATION	11
2.2.1 MODEL COMPLIANCE VERIFICATION SUMMARY.....	12
2.3 THE JPZZ APPROACH TO MODEL COMPLIANCE VERIFICATION.....	13
2.3.1 TOOL SUPPORT.....	15
2.4 RELEVANT MODELING LANGUAGES/NOTATIONS.....	16
2.4.1 QUERY/VIEW/TRANSFORMATION (QVT)	16
2.4.1.1 QVT Motivation	16
2.4.1.2 QVT Concepts	18
2.4.1.3 QVT as Standard Transformation Language.....	22
2.4.1.4 QVT Further Reading.....	28
2.4.2 OBJECT CONSTRAINT LANGUAGE.....	28
2.4.2.1 OCL Further Reading	31
2.4.3 UNIFIED MODELING LANGUAGE (UML).....	31
2.4.3.1 UML Further Reading.....	34
2.4.4 IBM RATIONAL ROSE REAL-TIME.....	35
2.4.4.1 Capsules.....	35
2.4.4.2 Capsule Roles	38
2.4.4.3 Protocols	38
2.4.4.4 Ports.....	39
2.4.4.5 Connectors	40
2.4.4.6 RoseRT Further Reading.....	40
2.5 A COMBINATORIAL TEST PATTERN	40

2.6	MODEL-TO-MODEL TRANSFORMATION TOOLS	41
CHAPTER 3		44
TOOL SUPPORT FOR MODEL COMPLIANCE VERIFICATION: USING QVT44		
3.1	A TOOL TO SUPPORT MODEL COMPLIANCE VERIFICATION	44
3.1.1	A PROTOTYPE TOOL FOR MODEL COMPLIANCE VERIFICATION	44
3.1.1.1	The Prototype	46
3.1.2	SUPPORTING MODEL COMPLIANCE VERIFICATION	57
3.1.2.1	Transforming a specification model to a tSpec model.....	57
3.1.2.2	Compliance Verification from tSpec to Spec.....	69
3.1.2.3	Compliance Verification from Design to tSpec	72
3.1.3	REVISITING OUR OBJECTIVES	77
3.2	COMPARISON TO AN EXISTING MODEL COMPLIANCE VERIFICATION TOOL....	79
3.2.1	THE MCV TOOL.....	79
3.2.2	PROTOTYPE VS. MCV: A COMPARISON	80
3.3	ADDRESSING THE CHALLENGES.....	82
3.3.1	REPRESENTATION LANGUAGE	83
3.3.2	TYPE CORRESPONDENCE.....	84
3.3.3	NAMING AND ROLE PLAYING	87
3.3.4	MANDATORY VS. OPTIONAL ELEMENTS	90
3.3.5	DESIGNS IMPLEMENTING A SUBSET OF A SPECIFICATION	93
3.3.6	MULTIPLE-ROLES.....	95
3.3.7	DESIGN IMPLEMENTING MORE THAN ONE SPECIFICATION	97
3.3.8	PACKAGE STRUCTURE	98
CHAPTER 4		102
CASE STUDY		102
4.1	CASE STUDY OVERVIEW	103
4.2	METAMODELS	104
4.2.1	UML 1.5	105
4.2.1.1	Attribute	106
4.2.1.2	Operation	106
4.2.1.3	Class	106
4.2.1.4	Interface	106
4.2.1.5	Package.....	107
4.2.1.6	Generalization.....	107
4.2.1.7	Association	107
4.2.1.8	AssociationEnd	107
4.2.1.9	Parameter	108
4.2.1.10	Interaction.....	108
4.2.1.11	Message	108
4.2.2	ROBERT.....	109
4.2.2.1	Attribute	109
4.2.2.2	Parameter	109
4.2.2.3	Operation.....	109
4.2.2.4	Class	110

4.2.2.5	Capsule	110
4.2.2.6	LogicalPackage.....	110
4.2.2.7	AssociationEnd	111
4.2.2.8	Association	111
4.2.2.9	Generalization.....	111
4.2.2.10	Protocol.....	112
4.2.2.11	CapsuleStructure.....	112
4.2.2.12	Connector.....	112
4.2.2.13	CapsuleRole.....	112
4.2.2.14	Port.....	113
4.2.2.15	Signal.....	113
4.3	MAPPING DEFINITIONS	113
4.3.1	ASSOCIATIONENDSTOPORTNAME	119
4.3.1.1	QVT Mapping Definition.....	119
4.3.2	ASSOCIATIONENDSTOPROTOCOLNAME.....	119
4.3.2.1	QVT Mapping Definition.....	120
4.3.3	ASSOCIATIONENDTOASSOCIATIONEND	120
4.3.3.1	QVT Mapping Definition.....	120
4.3.4	ASSOCIATIONTOASSOCIATION.....	121
4.3.4.1	QVT Mapping Definition.....	121
4.3.5	ASSOCIATIONTOPORT	122
4.3.5.1	QVT Mapping Definition.....	122
4.3.6	ASSOCIATIONTOPROTOCOL.....	123
4.3.6.1	QVT Mapping Definition.....	123
4.3.7	ATTRIBUTEATTRIBUTE	123
4.3.7.1	QVT Mapping Definition.....	124
4.3.8	CLASSTOCAPSULE	124
4.3.8.1	QVT Mapping Definition.....	124
4.3.9	CLASSTODATACLASS.....	125
4.3.9.1	QVT Mapping Definition.....	125
4.3.10	COLLECTPROTOCOLS.....	126
4.3.10.1	QVT Mapping Definition.....	126
4.3.11	GENERALIZATIONTOGENERALIZATION	127
4.3.11.1	QVT Mapping Definition.....	127
4.3.12	GENERALIZATIONTOPORT	127
4.3.12.1	QVT Mapping Definition.....	127
4.3.13	INTERACTIONADDTOPROTOCOL.....	128
4.3.13.1	QVT Mapping Definition	128
4.3.14	INTERACTIONTODATACLASSES.....	129
4.3.14.1	QVT Mapping Definition	129
4.3.15	INTERFACEOPERATIONTOSIGNAL	130
4.3.15.1	QVT Mapping Definition	130
4.3.16	INTERFACETODATACLASSES.....	131
4.3.16.1	QVT Mapping Definition	131
4.3.17	INTERFACETOPORTNAME.....	131
4.3.17.1	QVT Mapping Definition	131

4.3.18	INTERFACEToPROTOCOL.....	132
4.3.18.1	QVT Mapping Definition	132
4.3.19	MESSAGEToSIGNAL	133
4.3.19.1	QVT Mapping Definition	133
4.3.20	OPERATIONRESULTToSIGNAL.....	134
4.3.20.1	QVT Mapping Definition	134
4.3.21	OPERATIONToDATACLASS.....	135
4.3.21.1	QVT Mapping Definition	135
4.3.22	OPERATIONToOPERATION	136
4.3.22.1	QVT Mapping Definition	136
4.3.23	OPERATIONToSIGNAL	137
4.3.23.1	QVT Mapping Definition	137
4.3.24	PACKAGEToLOGICALPACKAGE.....	138
4.3.24.1	QVT Mapping Definition	138
4.3.25	PARAMETERToATTRIBUTE.....	139
4.3.25.1	QVT Mapping Definition	139
4.3.26	PARAMETERToPARAMETER	140
4.3.26.1	QVT Mapping Definition	140
4.3.27	PROCESSINTERACTIONS	140
4.3.27.1	QVT Mapping Definition	140
4.3.28	UMLToROSET	141
4.3.28.1	QVT Mapping Definition	141
4.4	MAPPING VERIFICATION.....	142
4.4.1	VERIFYING INTERFACEToPROTOCOL	142
4.4.2	VERIFYING ASSOCIATIONENDToASSOCIATIONEND.....	147
4.4.3	VERIFYING REMAINING MAPPINGS.....	150
4.5	COMPLIANCE RULES	151
4.5.1	ASSOCIATIONCOMPLIANCE	152
4.5.1.1	QVT Relation Definition	152
4.5.2	ASSOCIATIONENDCOMPLIANCE	152
4.5.2.1	QVT Relation Definition	152
4.5.3	ATTRIBUTECOMPLIANCE	153
4.5.3.1	QVT Relation Definition	153
4.5.4	CAPSULECOMPLIANCE	153
4.5.4.1	QVT Relation Definition	154
4.5.5	CAPSULEROLECOMPLIANCE	154
4.5.5.1	QVT Relation Definition	155
4.5.6	CAPSULESTRUCTURECOMPLIANCE.....	155
4.5.6.1	QVT Relation Definition	155
4.5.7	CLASSCOMPLIANCE.....	156
4.5.7.1	QVT Relation Definition	156
4.5.8	CONNECTORCOMPLIANCE	156
4.5.8.1	QVT Relation Definition	157
4.5.9	GENERALIZATIONCOMPLIANCE.....	157
4.5.9.1	QVT Relation Definition	157
4.5.10	OPERATIONCOMPLIANCE	157

4.5.10.1 QVT Relation Definition	158
4.5.11 PACKAGECOMPLIANCE.....	158
4.5.11.1 QVT Relation Definition	159
4.5.12 PARAMETERCOMPLIANCE.....	160
4.5.12.1 QVT Relation Definition	160
4.5.13 PORTCOMPLIANCE.....	160
4.5.13.1 QVT Relation Definition	160
4.5.14 PROTOCOLCOMPLIANCE.....	161
4.5.14.1 QVT Relation Definition	161
4.5.15 SIGNALCOMPLIANCE.....	162
4.5.15.1 QVT Relation Definition	162
4.6 COMPLIANCE RULE VERIFICATION.....	162
4.6.1 VERIFYING THE ATTRIBUTE COMPLIANCE RULE	163
4.6.2 VERIFYING THE CLASS COMPLIANCE RULE	164
4.7 EVALUATION OF COMPLIANCE	168
4.7.1 SPECIFICATION MODEL	168
4.7.2 DESIGN MODEL.....	169
4.7.3 GENERATED TSPEC MODEL.....	170
4.7.4 VERIFYING DESIGN COMPLIANCE	176
CHAPTER 5	179
CONCLUSION AND FUTURE WORK.....	179
5.1 FUTURE WORK.....	180
5.1.1 PROTOTYPE FUTURE WORK.....	180
5.1.2 QVT RELATED FUTURE WORK.....	181
5.2 CONTRIBUTIONS	179
5.3 CONCLUSION	181
REFERENCES	186
APPENDIX A.....	188
TABLE OF ACRONYMS	188
APPENDIX B.....	189
TOOL SCREEN CAPTURE.....	189
APPENDIX C.....	190
UML AND ROSE REAL-TIME METAMODELS	190
C.1 UML METAMODEL.....	190
C.2 ROSE RT METAMODEL.....	195
APPENDIX D	205
XML SCHEMAS.....	205
D.1 MODEL FILES.....	205
APPENDIX E.....	207
CASE STUDY : SPECIFICATION MODEL.....	207

APPENDIX F	223
CASE STUDY : DESIGN MODEL	223
F.1 BANKSYSTEM PACKAGE.....	223
F.1.1 PACKAGES.....	223
F.1.2 CLASSES.....	223
F.1.3 CAPSULES.....	223
F.1.4 PROTOCOLS	224
F.2 ACTIVECLASSES PACKAGE.....	224
F.2.1 PACKAGES.....	224
F.2.2 CLASSES.....	224
F.2.3 CAPSULES.....	225
F.2.4 PROTOCOLS	231
F.2.5 ASSOCIATIONS.....	237
F.3 BASETYPES PACKAGE.....	237
F.3.1 PACKAGES.....	237
F.3.2 CLASSES.....	237
F.3.3 CAPSULES.....	238
F.3.4 PROTOCOLS	238
F.3.5 ASSOCIATIONS.....	238
F.4 DATACLASSES PACKAGE.....	238
F.4.1 PACKAGES.....	238
F.4.2 CLASSES.....	239
F.4.3 CAPSULES.....	241
F.4.4 PROTOCOLS	241
F.4.5 ASSOCIATIONS.....	241
F.5 INTERFACES PACKAGE	242
F.5.1 PACKAGES.....	242
F.5.2 CLASSES.....	242
F.5.3 CAPSULES.....	244
F.5.4 PROTOCOLS	244
F.5.5 ASSOCIATIONS.....	245
F.6 SEQUENCEDIAGRAMS PACKAGE.....	245
F.6.1 PACKAGES.....	245
F.6.2 CLASSES.....	246
F.6.3 CAPSULES.....	248
F.6.4 PROTOCOLS	248
F.6.5 ASSOCIATIONS.....	248
APPENDIX G.....	249
CASE STUDY : TRACEABILITY MODEL EXTRACTS.....	249
G.1 UMLToROBERT	249
G.2 PROCESSINTERACTIONS.....	249
G.3 PARAMETERToPARAMETER	249
G.4 PARAMETERToATTRIBUTE	250
G.5 PACKAGEToLOGICALPACKAGE.....	250
G.6 OPERATIONToSIGNAL	250

G.7 OPERATIONTOOPERATION	250
G.8 OPERATIONTOCLASS	251
G.9 OPERATIONRESULTTO SIGNAL	251
G.10 MESSAGE TO SIGNAL	251
G.11 INTERFACE TO PROTOCOL	251
G.12 INTERFACE TO PORT NAME	252
G.13 INTERFACE TO DATA CLASSES	252
G.14 INTERFACE OPERATION TO SIGNAL	252
G.15 INTERACTION TO DATA CLASSES	253
G.16 INTERACTION ADD TO PROTOCOL	253
G.17 GENERALIZATION TO PORT	253
G.18 GENERALIZATION TO GENERALIZATION	253
G.19 COLLECT PROTOCOLS	254
G.20 CLASS TO DATA CLASS	255
G.21 CLASS TO CAPSULE	255
G.22 ATTRIBUTE TO ATTRIBUTE	255
G.23 ASSOCIATION TO PROTOCOL	256
G.24 ASSOCIATION TO PORT	256
G.25 ASSOCIATION TO ASSOCIATION	256
G.26 ASSOCIATION ENDS TO PROTOCOL NAME	256
G.27 ASSOCIATION ENDS TO PORT NAME	257
G.28 ASSOCIATION END TO ASSOCIATION END	257

LIST OF TABLES

<i>Number</i>	<i>Page</i>
TABLE 1 BRIEF EXPLANATION OF THREE MDA MODEL TYPES.....	2
TABLE 2 OCL OPERATORS THAT ARE RELEVANT TO THIS DISSERTATION.....	30
TABLE 3 OCL COLLECTION OPERATIONS THAT ARE RELEVANT TO THIS DISSERTATION.....	30
TABLE 4 OCL COLLECTION ITERATORS THAT ARE RELEVANT TO THIS DISSERTATION	31
TABLE 5 DEFINITION OF RELATION EVALUATION FAILURE TRACEABILITY MODEL ELEMENTS	56
TABLE 6 MAPPING APPLICATION VERIFICATION COMBINATORIAL MODEL, WHERE EACH ROW REPRESENTS A SEPARATE TEST CASE.....	63
TABLE 7 EXPLANATION OF SYMBOLS USED IN TABLE 6	63
TABLE 8 EXAMPLE OF METAMODEL ELEMENT EXPLANATION FOR THE PACKAGE ELEMENT IN FIGURE 69	105
TABLE 9 RELATIONSHIP BETWEEN UML 1.5 PACKAGE, CLASS, INTERFACE AND ASSOCIATION AND ROSE RT LOGICAL PACKAGE, CLASS, CAPSULE, PROTOCOL AND ASSOCIATION ..	114
TABLE 10 CHOICES FOR FEATURES THAT AFFECT THE INTERFACE TO PROTOCOL MAPPING	143
TABLE 11 COMBINATORIAL MODEL FOR VERIFYING THE INTERFACE TO PROTOCOL MAPPING	144
TABLE 12 EXPLANATION OF SYMBOLS USED IN TABLE 11	145
TABLE 13 ASSOCIATION END FEATURES AND THEIR CHOICES.....	147
TABLE 14 COMBINATORIAL MODEL DEFINING TEST CASES FOR VERIFYING THE ASSOCIATION END TO ASSOCIATION END MAPPING.....	148
TABLE 15 COMBINATORIAL MODEL DEFINING TEST CASES FOR VERIFYING THE ASSOCIATION END TO ASSOCIATION END MAPPING (CONTINUED)	149
TABLE 16 EXPLANATION OF SYMBOLS USED IN TABLE 14.....	149
TABLE 17 COMBINATORIAL MODEL THAT DEFINES THE TEST CASES FOR VERIFYING THE ATTRIBUTE COMPLIANCE RULE	163
TABLE 18 EXPLANATION OF SYMBOLS USED IN TABLE 16.....	163
TABLE 19 SET TESTING COMBINATIONS. ROWS MARKED ILLEGAL ARE DUE TO THE DEFINITION OF THE PROPER SUBSET OPERATOR	165
TABLE 21 COMBINATORIAL MODEL FOR TESTING THE CLASS COMPLIANCE RELATION. THE SYMBOL ‘t’ REPRESENTS tSPEC AND ‘d’ DESIGN. A CHECKMARK SIGNIFIES A SUCCESS AND AN X A FAILURE	166
TABLE 22 COMBINATORIAL MODEL FROM TABLE 19 CONTINUED	167
TABLE 23 SPECIFICATION MODEL ELEMENT TYPES AND THE NUMBER OF OCCURRENCES IN THE MODEL.....	169
TABLE 24 CASE STUDY MAPPING APPLICATION STATISTICS	172
TABLE 25 COMPLETENESS ANALYSIS ARTEFACT.....	173

LIST OF FIGURES

<i>Number</i>	<i>Page</i>
FIGURE 1 TWO-STEP APPROACH TO MODEL COMPLIANCE VERIFICATION [70].	14
FIGURE 2 A SIMPLE BANK MODEL	18
FIGURE 3 RESULT OF EVALUATING THE QUERY 'RETURN ALL OBJECTS OF TYPE CLASS WHOSE NAME BEGINS WITH B' ON THE BANK MODEL (FIGURE 2)	19
FIGURE 4 ENTITY-RELATIONSHIP VIEW OF THE BANK ACCOUNT MODEL (FIGURE 2)	20
FIGURE 5 MAPPING EXPLAINING THE PATTERN EXPRESSION AND CONDITION EXPRESSION	23
FIGURE 6 EXSMPL E OF A FREE VARIABLE THAT APPEARS IN TWO DOMAINS	24
FIGURE 7 SYNTAX OF A QVT RELATION DEFINITION	24
FIGURE 8 SYNTAX OF A QVT MAPPING DEFINITION	26
FIGURE 9 QVT MAPPING DEFINITION EXAMPLE WITH AN OBJECT EXPRESSION IN THE BODY	27
FIGURE 10 SEQUENCE DIAGRAM EXAMPLE	34
FIGURE 11 CAPSULE A'S EXTERNAL VIEW	36
FIGURE 12 CAPSULE A'S INTERNAL VIEW	37
FIGURE 13 PROTOCOL EXAMPLE	39
FIGURE 14 OVERVIEW OF THE TOOLS INDIVIDUAL COMPONENTS	46
FIGURE 15 OVERVIEW OF THE COMPLIANCE VERIFICATION IDE COMPONENTS	47
FIGURE 16 THE ELEMENTS OF A MODEL COMPLIANCE PROJECT	48
FIGURE 17 A BANK SYSTEM MODEL USING A SIMPLIFIED UML LANGUAGE	50
FIGURE 18 SCREEN CAPTURE OF THE MODEL INSPECTOR AFTER LOADING THE BANKSYSTEM (FIGURE 17)	51
FIGURE 19 THE GENERAL FORMAT OF TRACEABILITY ITEMS	53
FIGURE 20 EXAMPLE OF A TRACEABILITY MODEL ELEMENT	54
FIGURE 21 THE GENERAL FORM OF MAPPING STATISTICS PRODUCED BY QVT ENGINE	54
FIGURE 22 EXAMPLE, A SUBSET OF THE MAPPING STATS PRODUCED FOR THE EVALUATION OF UMLToROSERT ON A MODEL	55
FIGURE 23 RELATION FAILURE TRACEABILITY ITEM GENERAL FORM	55
FIGURE 24 EXAMPLE OF A RELATION FAILURE TRACEABILITY MODEL	56
FIGURE 25 UMLCLASSToTABLE.QVT MAPPING DEFINITION	58
FIGURE 26 SCREEN CAPTURE OF MAPPINGS FOR UML TO DATABASE MODEL EXAMPLE	59
FIGURE 27 EXAMPLE METAMODELS	60
FIGURE 28 MAPPING TO BE VERIFIED FOR CORRECT APPLICATION	62
FIGURE 29 DRIVER FOR VERIFYING MAPPING IN FIGURE 28	62
FIGURE 30 PACKAGEMapping MAPPING FOR COMPLETENESS VERIFICATION EXAMPLE	64
FIGURE 31 CLASSMapping MAPPING FOR COMPLETENESS VERIFICATION EXAMPLE	65
FIGURE 32 ATTRIBUTEMapping MAPPING FOR COMPLETENESS VERIFICATION EXAMPLE	65
FIGURE 33 A UML MODEL FOR COMPLETENESS TESTING EXAMPLE	65
FIGURE 34 TRACEABILITY INFORMATION PRODUCED WHEN APPLYING THE MAPPING IN FIGURE 29 TO THE MODEL IN FIGURE 33	66
FIGURE 35 CONTINUATION OF THE TRACEABILITY INFORMATION PRODUCED WHEN APPLYING THE MAPPING IN FIGURE 29 TO THE MODEL IN FIGURE 33	67

FIGURE 36 RESULTS FROM THE AUTOMATED ANALYSIS OF FOR COMPLETENESS.....	68
FIGURE 37 MAPPING WHERE ATTRIBUTE ELEMENTS ARE USED BUT WOULD NOT BE REPORTED IN THE TRACEABILITY MODEL	69
FIGURE 38 MAPPING DEFINITION THAT PRODUCES A CSHARPClass FROM A UMLClass	70
FIGURE 39 REDEFINITION OF THE MAPPING DEFINED IN FIGURE 28.	71
FIGURE 40 RULE FOR VERIFYING COMPLIANCE BETWEEN INPUT AND OUTPUT OF THE UMLClassToCSHARPClass MAPPING, SUPPORTED BY THE RELATIONS IN FIGURE 41 AND FIGURE 42	71
FIGURE 41 RULE FOR COMPLIANCE BETWEEN A UMLAttribute AND A CSHARPAttribute.....	72
FIGURE 42 RULE FOR COMPLIANCE BETWEEN A UMLAttribute AND CSHARPProperty ...	72
FIGURE 43 A DESIGN THAT IS COMPLIANT WITH A TSPEC	74
FIGURE 44 A DESIGN THAT DOES NOT COMPLY WITH THE TSPEC.....	75
FIGURE 45 RULE DEFINING COMPLIANCE BETWEEN TWO CSHARPClass CLASSES	75
FIGURE 46 RULE DEFINING COMPLIANCE BETWEEN TWO CSHARPAttribute CLASSES.....	76
FIGURE 47 RULE DEFINING COMPLIANCE BETWEEN TWO CSHARPProperty CLASSES	76
FIGURE 48 EXAMPLE OF RESULTS PRODUCED WHEN COMPLIANCE FAILS.....	76
FIGURE 49 TYPE CORRESPONDENCE EXAMPLE	84
FIGURE 50 EXAMPLE OF THE ADDITIONAL INFORMATION MODEL FOR SOLUTION TO TYPE CORRESPONDENCE CHALLENGE.....	85
FIGURE 51 MAPPING A CLASS TO A CLASS INCORPORATING THE TYPE CORRESPONDENCE INFORMATION	86
FIGURE 52 MAPPING AN ATTRIBUTE TO AN ATTRIBUTE MAKING USE OF THE TYPE CORRESPONDENCE INFORMATION.....	86
FIGURE 53 TYPE CORRESPONDENCE EXAMPLE COMPLIANCE RULES	87
FIGURE 54 NAMING AND ROLE-PLAYING EXAMPLE.....	87
FIGURE 55 EXAMPLE OF THE ADDITIONAL INFORMATION MODEL FOR SOLUTION TO NAME AND ROLE PLAYING CHALLENGE	88
FIGURE 56 MAPPING A CLASS TO A CLASS INCORPORATING THE NAME CORRESPONDENCE INFORMATION	89
FIGURE 57 MAPPING AN ATTRIBUTE TO AN ATTRIBUTE MAKING FOR THE NAME CORRESPONDENCE EXAMPLE	89
FIGURE 58 MANDATORY VS. OPTIONAL EXAMPLE.....	90
FIGURE 59 EXAMPLE OF ADDITIONAL INFORMATION USED IN ADDRESSING THE OPTIONAL ELEMENT CHALLENGE.....	91
FIGURE 60 MAPPING THAT ONLY MAPS OPTIONAL CLASS ELEMENTS THAT HAVE BEEN MARKED AS REALIZED	92
FIGURE 61 A COMPLIANT DESIGN TAKING INTO ACCOUNT THE OPTIONAL SPECIFICATION ELEMENTS NOT REALIZED.....	92
FIGURE 62 IMPLEMENTING A SUBSET OF THE SPECIFICATION.....	94
FIGURE 63 EXAMPLE OF AN ADDITIONAL INFORMATION MODEL FOR RESOLVING THE SUBSET CHALLENGE.....	95
FIGURE 64 EXAMPLE OF ADDITIONAL INFORMATION FOR THE MULTIPLE-ROLE CHALLENGE.....	96
FIGURE 65 DECISION PROCESS FOR VERIFYING COMPLIANCE WITH MULTIPLE SPECIFICATIONS	98
FIGURE 66 PACKAGE STRUCTURE EXAMPLE	99

FIGURE 67 EXAMPLE TSPEC MODEL AND DESIGN MODEL FOR PACKAGE STRUCTURE	
CHALLENGE	100
FIGURE 68 COMPLIANCE RULE THAT IGNORES THE PACKAGE STRUCTURE IN THE MODEL...	101
FIGURE 69 METAMODEL FOR MYUML MODELING LANGUAGE	104
FIGURE 70 MAPPING AN ACTIVE ASSOCIATION	116
FIGURE 71 MAPPING A MESSAGE	116
FIGURE 72 MAPPING A GENERALIZATION BETWEEN A INTERFACE AND AN ACTIVE CLASS	117
FIGURE 73 UML 1.5 TO ROSEXT MAPPING DEPENDENCY GRAPH	118
FIGURE 74 TEST DRIVER FOR TESTING THE INTERFACETOPROTOCOL MAPPING	146
FIGURE 75 EXAMPLE OF AN INTERFACE ELEMENT THAT CORRESPONDS TO ROW 5 OF TABLE	
11	146
FIGURE 76 INSTANTIATION OF TEST CASES 15 AND 25 FROM TABLE 14	150
FIGURE 77 TEST DRIVER FOR THE ASSOCIATIONENDTOASSOCIATIONEND MAPPING	150
FIGURE 78 ROSEXT DESIGN TO ROSEXT TSPEC COMPLIANCE RULE DEPENDENCY GRAPH	151
FIGURE 79 ATTRIBUTE COMPLIANCE RULE TEST DRIVER	164
FIGURE 80 A TEST DRIVER FOR TESTING THE CLASS COMPLIANCE RELATION AGAINST A SET OF	
TEST	167
FIGURE 81 COMPLIANCE FAILURE TRACEABILITY MODEL PRODUCED WHEN DESIGN IS MISSING	
THE BASETYPES PACKAGE	177
FIGURE 82 NON-COMPLIANCE INFORMATION PRODUCED WHEN DESIGN IS MISSING THE	
CREDITRATING ATTRIBUTE IN THE CUSTOMER CLASS	178
FIGURE 83 PROTOTYPE SCREEN CAPTURE	189
FIGURE 84 AUXILLARY ELEMENTS	191
FIGURE 85 BACKBONE ELEMENTS	192
FIGURE 86 CLASSIFIERS	193
FIGURE 87 RELATIONSHIP ELEMENTS	194
FIGURE 88 ASSOCIATION ELEMENTS DIAGRAM	195
FIGURE 89 ROSEXT ATTRIBUTE ELEMENTS DIAGRAM	196
FIGURE 90 ROSEXT CAPSULE ELEMENT DIAGRAM	197
FIGURE 91 ROSEXT CLASS ELEMENTS DIAGRAM	198
FIGURE 92 ROSEXT CLASSIFIERROLE ELEMENTS DIAGRAM	199
FIGURE 93 ROSEXT COLLABORATION ELEMENTS DIAGRAM	199
FIGURE 94 ROSEXT CONNECTOR ELEMENT DIAGRAM	200
FIGURE 95 ROSEXT LOGICALPACKAGE ELEMENTS DIAGRAM	201
FIGURE 96 ROSEXT OPERATION ELEMENTS DIAGRAM	202
FIGURE 97 ROSEXT PROTOCOL ELEMENTS DIAGRAM	202
FIGURE 98 ROSEXT RELATION ELEMENTS DIAGRAM	203
FIGURE 99 ROSEXT CLASSIFIER ELEMENTS DIAGRAM	204
FIGURE 100 BANK SYSTEM SPECIFICATION MODEL PACKAGES	207
FIGURE 101 SPECIFICATION MODEL BASETYPES PACKAGE	207
FIGURE 102 SERVER AND MANGER ELEMENTS OF THE SPECIFICATION MODEL ACTIVECLASSES	
PACKAGE	208
FIGURE 103 SPECIFICATION MODEL ACTIVECLASSES	
PINVALIDATIONTRANSACTIONMANAGER DIAGRAM	209
FIGURE 104 SPECIFICATION MODEL ACTIVECLASSES QUERYTRANSACTIONMANAGER	
DIAGRAM	210

FIGURE 105 SPECIFICATION MODEL ACTIVECLASSES TRANSFERTRANSACTIONMANAGER DIAGRAM	211
FIGURE 106 SPECIFICATION MODEL ACTIVECLASSES WITHDRAWALTRANSACTIONMANAGER DIAGRAM	212
FIGURE 107 SPECIFICATION MODEL DATACLASSES PACKAGE	213
FIGURE 108 SPECIFICATION MODEL DATACLASSES PACKAGE AUXILIARY ELEMENTS	214
FIGURE 109 ADDINTEREST SEQUENCE DIAGRAM	214
FIGURE 110 CHECKINGTOCHECKINGTRANSFER SEQUENCE DIAGRAM	215
FIGURE 111 CHECKINGTOSAVINGS SEQUENCE DIAGRAM	215
FIGURE 112 CHECKINGWITHDRAWFUNDS SEQUENCE DIAGRAM	216
FIGURE 113 CLOSECHECKINGACCOUNT SEQUENCE DIAGRAM	216
FIGURE 114 CLOSESAVINGSACCOUNT SEQUENCE DIAGRAM	217
FIGURE 115 INITIALIZE SEQUENCE DIAGRAM	217
FIGURE 116 OPENCHECKINGACCOUNT SEQUENCE DIAGRAM	218
FIGURE 117 OPENSAVINGSACCOUNT SEQUENCE DIAGRAM	218
FIGURE 118 QUERYCHECKINGACCOUNT SEQUENCE DIAGRAM	219
FIGURE 119 QUERYSAVINGSACCOUNT SEQUENCE DIAGRAM	219
FIGURE 120 SAVINGSTOCHECKINGTRANSFER SEQUENCE DIAGRAM	220
FIGURE 121 SAVINGSTOSAVINGSTRANSFER SEQUENCE DIAGRAM	220
FIGURE 122 SAVINGSWITHDRAWFUNDS SEQUENCE DIAGRAM	221
FIGURE 123 TRANSFERFUNDSABORT SEQUENCE DIAGRAM	221
FIGURE 124 WITHDRAWFUNDSABORT SEQUENCE DIAGRAM	222

Chapter 1

INTRODUCTION

The introduction establishes the context and motivation for our research by presenting a brief look at modeling and model compliance verification. The problem, namely providing tool support for model compliance verification is explicitly stated. The core objective of the research along with several sub-objectives and contributions are presented. Finally, an overview of the remaining chapters completes the introduction.

1.1 Context and Motivation

Three different aspects of software engineering drive this research. The first is the use of models in the software development process, and more specifically models that capture a specification (e.g. Software Communications Architecture (SCA) [27] or Software Radio Components Specification (SRC) [44]) and models that capture a *design*. The second aspect, is verifying a *design* model for compliance with a *specification* model. Finally, the third aspect is the need for tools to support the activities in software development, more specifically in the area of model driven development [52]. We briefly explore each of these in the following subsections.

1.1.1 Models

A software model is understood to be an abstraction of a system [17]. That is, a model suppresses the irrelevant details and focuses on and/or emphasizes the most relevant details. The increase in software system complexity has driven the need for more abstract models than the code model. For example, the architectural model of a system emphasizes the components, ports and connectors of a system and suppresses system state.

The introduction of the Model Driven Development (MDD) [52] initiative has illustrated the importance of models in the software development process. The MDD initiative proposes that models of a higher-level of abstraction be transformed into models that are more detailed. The more detailed models may in fact contain code or be the code.

The Object Management Group's (OMG) Model Driven Architecture (MDA) [18, 29, 32, 37] defines a framework of standards to enable MDD. This framework includes the Meta-Object Facility (MOF) standard [42], the Unified Modeling Language (UML) [38, 40, 45] standards and the MOF 2.0 Query/View/Transformation (QVT) standard [36]. The MDA framework also defines different types or classifications of models as shown in Table 1.

Table 1 Brief explanation of three MDA model types

Model Type	Explanation
Computation Independent Model (CIM)	A model that is independent of computational technology.
Platform Independent Model (PIM)	A model that is independent of a particular platform.
Platform Specific Model (PSM)	A model that is dependent upon a particular platform.

The OMG's MOF 2.0 Query/View/Transformation (QVT) standardization effort [36] is of particular interest to this dissertation. The QVT is currently a work in progress; that is the final *specification* has not been adopted by the OMG. However, once the QVT standard is complete, it will provide a standardized language for specifying queries, views and transformations on models. The motivation behind the QVT standard is current model manipulation and transformation technologies are vendor and/or tool specific. This vendor specific approach requires that model-to-model transformations be re-implemented or re-written to adhere to a vendors' technology or language.

1.1.2 Standard specification – product development compliance

This subsection considers the activities pertaining to the use of a *specification* expressed in the form of a model from which future software products will be developed and ultimately verified. We take a *specification* to be the definition of the required properties and capabilities of a system or

family of systems within a particular domain. That is, the *specification* defines the properties, capabilities and responsibilities that all realizations of the *specification* must exhibit. In this dissertation we restrict ourselves to *specifications* expressed as a model. Additionally, the modeling language or notation must have a well defined metamodel.

Within MDD (and MDA as it is a specialization of MDD) most implementations are expressed as models. Multiple models at different levels of abstraction may be used in the development of a particular system or family of systems. Each of the models may use a different modeling language. The MDA defines three classes of models, Computation Independent Models (CIMs), Platform Independent Models (PIMs) and Platform Specific Models (PSMs), where usually the PSMs are at a lower level of abstraction than the PIMs, which are less abstract than CIMs. In MDD/MDA, the executable implementation is usually generated or derived from a PSM, whereas the *specification* is a PIM. In this work we compare the *specification* (usually a PIM) with the model(s) (usually a PSM) that will eventually be used to generate implementations (also known as the *design* model(s)).

Therefore, as defined in [70] model compliance verification is the activity of comparing a *design* model to a *specification* model to ensure that it satisfies all of the requirements of the *specification* model. More specifically, the *design* model is analyzed to ensure that it conforms to all of the structural (properties) and behavioural (capabilities) requirements defined in the *specification* model. The authors of [70] demonstrate that the activities of structural and behavioural compliance verification are quite different. The approach to behavioural compliance verification they discuss relies on vendor specific testing tools and model executability. Thus, in this dissertation we focus on the structural aspect of compliance.

1.1.3 Tool Support

There has been substantial research into the activities that should be included in a software development process (e.g. [1, 3, 4, 33, 49]). The consensus is that for an activity to be practical in an industrial setting, tool support must exist for that activity. Many tools exist for the activities in the software development process, for example Telelogic DOORS [63] for requirements management and JUnit [28] for unit testing. Most important to this dissertation are software

modeling tools, such as IBM Rational Rose Enterprise [56], Microsoft Visio [9], Borland Together Designer [10], which can be used to model the *specification* and *design* of software systems. We rely on the existence of such modeling tools and more specifically the IBM Rational Rose Enterprise and IBM Rational Rose Real-Time [56] tools, which we use in our case study (Chapter 4).

One activity that has little or no tool support to our knowledge is that of model compliance verification [70]. That is, while tools such as Rational Rose provide the ability to model a software system, they do not provide the facilities to check a model for compliance with a second model. Tool support for this particular activity is the focus of this dissertation.

1.2 Problem Statement

The general question we consider is whether it is feasible to provide tool support for model compliance verification through the use of a standardized mapping and compliance rule definition language. The mappings and rules defined in this standardized language are used to verify systematically the compliance of a *design* model with a *specification* model. We restrict ourselves to models based on well-defined yet potentially different metamodels. In the context of this dissertation, the OMG's QVT language¹ is used as the standardized mapping and compliance rule language.

We claim that it is feasible to provide tool support for model compliance support, by showing that the QVT can be used to semi-automatically verify compliance between a *specification* expressed in standard UML and a *design* expressed in RoseRT. This is achieved through the development of a prototype tool and its use in a case study.

More generally, we make use of the QVT syntax and semantics to provide a prototype that is independent of modeling languages and that supports the activity of model compliance

---

¹ We note that the QVT standard is currently a work in progress. The submission by the QVT-Merge Group [31] is considered to be the standard.

verification. We follow closely the approach to model compliance verification outlined in [70]. To our knowledge no such tools exist.

1.3 Proposed Solution

Our proposed solution to showing the feasibility of tool support for model compliance is a prototype that uses the QVT transformation definition language for the definition of mappings and compliance rules. As will be discussed in Section 2.4.1, the QVT transformation language defines two types of transformations: mappings and relations. The proposed prototype is influenced by the model compliance verification process defined in [70]. More specifically, QVT mapping definitions are used to transform the *specification* model into an intermediate model in the modeling language used for the *design* model. QVT relation definitions express the semantics of compliance between the *design* model and the intermediate model.

The prototype proposed in this dissertation allows a software engineer to verify a system *design* specified in the IBM Rational Rose Real-Time toolset against a *specification* specified in the IBM Rational Rose toolset for compliance. The actual transformation and verification of the models, as described in the previous paragraph, is achieved by using the QVT [31] standard under development by the OMG.

In summary, we provide tool support for model compliance verification by creating a *specification* model or loading a *specification* model from a modeling tool (e.g. IBM Rational Rose), applying mappings, written in the QVT language, to it to generate an intermediate *specification* in the *design* language. This intermediate model is known as the *tSpec* model. The *design* model created within the prototype or loaded from a third-party modeling tool (e.g. Rational Rose Real-Time) is then verified against this intermediate *specification* model, through compliance rules written in the QVT language. The result of the process will indicate whether the *design* model complies with the *specification* model.

1.4 Validation of Proposed Solution

In this section, we address the validation of our proposed solution. As is common in software engineering dissertations we will make use of a case study. In [70] the Software Communications Architecture (SCA) specification [27] is used for the case study and also drives much of research context. The SCA is a large and complex standard specification² that contains many ill-formed elements (for example attributes names that contain spaces). Since the objectives of this dissertation are different from those of [70], namely we are interested in evaluating the feasibility of tool support for model compliance verification, we have chosen what we feel is a much simpler *specification* model, in the name of understandability. More specifically, we will use a fictitious specification for a Bank server system. The specification language (UML 1.5) modeling elements (for example Class, Interface, etc.) used in defining Bank server specification are influenced by those that appear in the SCA specification and other standard specifications. The case study defines a set of mappings necessary to transform a UML 1.5 *specification* model into a RoseRT *tSpec* model, as well as, the compliance rules to evaluate a Bank server *design* model for compliance with the *tSpec* model.

1.5 Contributions

We claim the following contributions:

- An initial demonstration that, through the use of a model-to-model transformation language it is possible to support the activity of model compliance verification. More specifically, we show that the OMG's QVT textual notation can be used to support model compliance verification.
- In principle, due to the use of QVT, more general support for the activity of compliance verification than defined in [70]. That is, the tools in [70] are specific to a

---

² A standard specification is defined by a standardization body or organization.

subset of UML 1.5 for the *specification* model and a subset of RoseRT for the *design* model.

- Definition of QVT mappings to transform a model using a subset UML 1.5 into a RoseRT. The subset of UML 1.5 is driven by the subset that is used in industrial specifications.
- Definition of QVT relations (rules for compliance) to analyze compliance in the context of a subset RoseRT.
- Show an approach to unit testing a QVT transformation (mapping or relation) using QVT mappings and relations.

1.6 Outline of Dissertation

The remainder of this dissertation is organized as follows:

Chapter 2 provides the necessary background knowledge for the understanding of the remaining chapters. This includes a review of the activity of model compliance verification, combinatorial models, the relevant languages and notations, and model-to-model transformation tools and prototypes. The languages and notations are the Unified Modeling Language (UML), IBM Rational Rose Real-Time notation, Object Constraint Language (OCL) and the Query/Views/Transformations (QVT) standard.

Chapter 3 discusses our proposed solution to the problem addressed by this dissertation. More specifically, a prototype for the activity of model compliance verification is presented. This is accompanied by a discussion of how it can support the activity of model compliance verification defined in [70]. The chapter concludes with an examination of some of the challenges in model compliance verification, along our prototype specific proposed solution.

Chapter 4 provides a case study to demonstrate the use of our proposed prototype for the activity of model compliance verification.

Chapter 5 provides a comprehensive summary of the work achieved. Then perspectives for further development and research are considered along with concluding remarks.

Appendix A provides a complete list of the acronyms used throughout this dissertation.

Appendix B contains a screen capture of the proposed prototype.

Appendix C contains the metamodels for the subset of both the UML 1.5 and RoseRT notation relevant to this dissertation.

Appendix D provides the XML schema for the models saved by the proposed prototype.

Appendix E contains the specification model for the case study used in this dissertation.

Appendix F contains the design model for the case study used in this dissertation.

Appendix G contains information necessary for the discussion of the case study in Chapter 5.

Chapter 2

BACKGROUND

This chapter provides an overview of the knowledge and technology that is relevant to this dissertation. This includes a discussion of a specification model vs. a design model and a summary of the activity of model compliance verification as defined in [70]. This is followed by a review of the modeling languages and notations that are relevant to this dissertation. The chapter concludes with a brief description of a combinatorial test pattern and other model-to-model transformation tools.

2.1 Specification vs. Design

In this section, we review what differentiates a *specification* from a *design*. More importantly, the differentiating factors and relationships that exist between *design* models and *specification* models. We first look at *designs* and *specifications* individually and conclude with the differences that exist between them.

2.1.1 Specification

The specification for a software system defines its required properties and capabilities, and the constraints on those properties and capabilities [48, 51, 57]. Specifications come in all shapes and sizes; textual, graphical model, formal, informal etc. The actual specification is either derived through discussion with system stakeholders and domain experts, from an industry specification, or a combination of the two.

Industry standard specifications are defined by Organizations such as the Object Management Group (OMG) [41], International Telecommunication Union (ITU) [26] and International

Organization for Standardization (ISO) [25] and other organizations must conform to these specifications when developing certain products. For example, if Company ABC is building a Software Defined Radio (SDR), they must adhere to the SCA standard [27]. From an abstract point of view developing a product that adheres to a *specification* requires transforming³ the *specification* into some form of *design* from which the eventual implementations will be derived.

2.1.2 Design

A *design* provides (or should provide) an unambiguous detailed definition of the structure and behaviour of the eventual implementation [48, 51, 57]. The *design* may contain assumptions that take into account the constraints imposed by the eventual operating environment, for example the hardware. More importantly, the *design* has more detail than a specification and drives the implementation of the final product.

In our research we restrict ourselves to *designs* that are documented as a model. Furthermore, the modeling language or notation used to document the *design* must have a well defined metamodel. A *design* model that uses the UML often contains more than one view or project of the system. For example, in the UML 2.0 composite structure diagrams and elements can be used to model the architecture of the system and state machines can be used to capture the behaviour.

2.1.3 Specification vs. Design

As pointed out in [70], several gaps exist between *specification* models and *design* models. The first being the level of abstraction used for the *specification* (generally a high-level of abstraction) and the one used for the *design* (generally a low-level of abstraction). The second gap being, that the *specification* modeling language often differs from the *design* modeling language. The gap in modeling languages is often a consequence of the difference in abstraction levels between a *specification* model and a *design* model. For a more complete discussion of the gaps between a

---

³ We recognize that such a transformation process may not be explicit.

specification model and a *design* model please see [70] : we deal with some of the gaps mentioned there in more detail in Section 3.3.

Intuitively, a *specification* model contains the 'what' and usually none of the 'how' whereas a *design* model contains a more detailed 'what' and usually most of the 'how'. Where 'what' defines the capabilities and properties of the system and the 'how' defines the manner in which these capabilities will be achieved. Finally, although not relevant to this dissertation the *implementation* should contain all of the 'what' and the entire 'how'.

2.2 Model Compliance Verification

In this section, we describe the activity we call “*model compliance verification*” and why it is important especially in the presence of a *specification* model. We take compliance to mean that an artefact conforms to the capabilities, properties and constraints defined in a *specification*. That is, if A is compliant with B then A satisfies all of the capabilities, elements and constraints of B. Verification is defined in [2] as ‘*Has the thing been built right?*’, namely, does the system exhibit the properties and behaviour that it claims it does.

In this dissertation, we are concerned with compliance between two models one being the *specification* model and the other being the *design* model. Using a simple example, we differentiate between traditional compliance as defined in the previous paragraph and model-to-model compliance.

Traditional compliance

Consider a class that realizes two interfaces each with one operation. If the class implements those two operations then it is deemed compliant with its interfaces. Furthermore, compliance testing is concerned with verifying that the artefact under test (AUT) conforms to its specification. For example, compliance tests for a software component will invoke the services of that component and verify that the exhibited behaviour conforms to the specification. That is, the

compliance tests demonstrate that the component complies with the specification that the component is realizing [2].

Model-to-model compliance

Consider, an active class that realizes two interfaces (each with one operation) defined in a *specification*. Suppose for example, this specified active class is realized in a *design* model as a capsule. We note that a capsule's only means of communication is through ports. In this case, compliance is not readily defined and must consider questions such as:

- It is a capsule that contains two ports each with one signal;
- It is a capsule with a single port with two signals; or
- It is a capsule with three ports (i.e. it has more capabilities than required in the *specification*).

Through this example it is evident that the activity of model compliance is not trivial. A software engineer must explicitly state the semantics of compliance between two or more models. That is, without additional information it is difficult, if not impossible, to determine which case(s) is(are) compliant.

2.2.1 Model Compliance Verification Summary

Given the independent concepts of model-to-model compliance and verification we define here the notion of model-to-model compliance verification. We take model-to-model compliance verification to be the activity of assessing whether a model at a lower or equal level of abstraction complies with a second model. More specifically, in the context of this dissertation we are concerned with assessing whether a *design* model complies with a *specification* model.

2.3 An Approach to Model Compliance Verification

In this section, we review the compliance verification approach put forth in [70]. Most, importantly we consider the need for a tool in which mappings and compliance rules can be easily defined, maintained, and modified. As previously described in Chapter 1 this last statement is the motivation behind this dissertation and more specifically the choice of the QVT technology.

In [70] the author defines a two-step approach for model compliance verification. In essence, the approach splits transformation and compliance activities between a *specification* model and a *design* model into two separate steps as illustrated in Figure 1. First, the original *specification* is transformed into a semantically equivalent⁴ model expressed in the language used to define the *design* model (design modeling language). This transformation bridges the semantic gap between the *specification* (or source) metamodel or modeling language and the *design* (or target) metamodel or modeling language. Second, a *design* model is to be obtained and its compliance to the transformed *specification* verified.

Activities defined in the approach include:

1. Transformation from *specification* model to *tSpec* model.
  - Semantic definition and validation of mappings.
  - Verification of correct application of mappings.
2. Compliance verification from *tSpec* model to *specification* model.
  - Traceability relationships from *tSpec* model to *specification* model.
  - Verification of a complete definition of mappings.

---

⁴ Semantic equivalence is defined and verified by a domain expert, and is not necessarily formally established.

Figure 1 Two-step approach to model compliance verification [70].

3. Transformation from *tSpec* model to *design* model.

- This particular activity is designer specific and out of the scope of the process defined in [70].
- In essence, a designer takes inspiration from the *specification* and/or its equivalent transformed *specification*, to manually produce a design.

4. Compliance verification from *design* model to *tSpec* model.

- Structural compliance – verification that a *design* model possesses the structural elements (along with their properties and relationships) as defined in the *specification* model (i.e., here the *tSpec*).
- Behavioural compliance – verification that element instances at the execution of a *design* model are capable of performing a sequence of message interchanges as defined in the *specification*.

More on the specifics of each activity can be found in [70]. Although the approach addresses proposals for both structural and behavioural compliance, this dissertation focuses only structural compliance between a *design* and its corresponding transformed *specification* (*tSpec*). We feel that the activities of structural compliance verification and behaviour compliance verification are different.

Additionally, the approach to behavioural compliance verification in [70] relies on model executability and a testing tool, IBM Rational Test Real-Time, which is specific to IBM Rational Rose Real-Time. From this viewpoint, we believe the challenges of behavioural compliance represent another significant endeavour by themselves.

Both the *specification* model and the *design* model(s) are inputs⁵ to the research; therefore, we avoid discussing purely (one-way) transformational approaches that go (via one or more correctness preserving transformations, e.g. [6, 7, 14]) from a *specification* model to its corresponding *design* model. That is, we address the issue of compliance between two input models, which is a quite different problem from proving that a *design/specification* model was correctly derived from a *specification/design* through a series of transformations.

2.3.1 Tool support

The process defined in [70] has tool support, however, each of the mappings and compliance rules is written in the C++ programming language and compiled. Therefore, changes to the mappings and rules require a modification to C++ source code and recompiling the tool. Furthermore, the tool described in [70] is restricted to *specification* models defined in the IBM Rational Rose toolset and *design* models defined in the IBM Rational Rose Real-Time toolset. Changing the *specification* or *design* tool would require substantial modifications to the tool. More specifics on this particular tool are provided in Section 3.2.1.

In this dissertation, we investigate tool support for model compliance verification that is tool independent, metamodel or modeling language independent and programming language independent. We remark, that although the *specification* modeling tool and *design* modeling tool used in this dissertation's case study are the same as that used in [70] our proposed prototype allows one to change either tool without changing the core of compliance verification prototype or the mapping and rule definition language. However, if the metamodel for either the *design* or *specification* changes it is likely that the mapping and/or compliance rules will have to be modified.

---

⁵ Supplied by a third-party.

2.4 Relevant Modeling Languages/Notations

In this section we review the modeling languages and notations that are relevant to this dissertation. For each language or notation the review is brief; we refer the readers interested in more details to the respective standards or books available. The languages reviewed are Query/View/Transformations (QVT), Object Constraint Language (OCL), the Unified Modeling Language (UML) and the Rational Rose Real-Time (RoseRT) toolset modeling language.

2.4.1 Query/View/Transformation (QVT)

In this section, we review the OMG's MOF 2.0 Query/Views/Transformations (QVT) standardization effort [36]. This includes the motivation behind the QVT standard.

The QVT is motivated by the need for a standardized approach for defining and executing model transformations and manipulating models, most importantly in the context of the MDA. The QVT Request for Proposal (RFP)⁶ [36] was first released in April of 2002. Initially the OMG received eight submissions to the QVT RFP, from which five submitted first revised proposals. Since the submission of the first revised proposal five of the submitters have consolidated into one submission known as QVT Merge [46]. In the remainder of this dissertation we focus on the QVT Merge submission [31], which appears to be the leading submission.

2.4.1.1 *QVT Motivation*

Each of the submissions in response to the QVT RFP is required to satisfy at least the mandatory requirements laid out in the RFP. The mandatory requirements included:

- Definition of a language for querying models;

---

⁶ Which will be referred to as the QVT RFP.

- Definition of a language for transformation definitions between models that may have different metamodels;
- Definition of a language for transformation definitions rich enough to express all information required to automatically generate the target model from the source model(s);
- Definition of a language for transformation definitions that allows for the creation of a view of a model(s);
- Definition of a language for transformation definitions that is declarative in order to support incremental changes in a source model(s);
- The abstract syntax of the query, view and transformation languages must be defined as a MOF 2.0 metamodel(s); and
- The QVT must operate on models defined using MOF 2.0.

We remark that the capabilities described in the QVT RFP and its submissions extend beyond the needs of this dissertation. That is, a QVT submission is supposed to satisfy a wide range of model transformations and queries, whereas our research use of a subset of the capabilities defined in the QVT RFP, namely the model transformation component.

The QVT submissions are subject to modifications until the final adopted specification is released. Therefore, it is possible that the transformation definition language may change, however, we do not feel that this will change the feasibility of using QVT in supporting model compliance verification, since the objectives outlined in the QVT RFP will not change. The remainder of this section defines the concepts as specified in the QVT RFP and QVT Merge submission.

2.4.1.2 QVT Concepts

Queries

A query is an expression that is evaluated over a model [19]. The result of such an evaluation is one or more instances of types defined in the source model or in the query language. For example, we could run a query over a UML source model to: *Return all objects of type Class whose name begins with 'B'.* If we ran such a query against the model presented in Figure 2, it would return those classes that are shaded in Figure 3: Bank, Branch and BranchManager.

Figure 2 A simple bank model

The metamodel of a view is not required to be the same as the metamodel of the base model. For example, Figure 4 presents an entity relationship view of the Simple Bank Account model in Figure 2. In this example, we have not created a new model only a projection of that model onto an entity-relational metamodel.

The QVT Merge submission defines a view using the mapping concept and thus does not require a separate view definition language. Therefore, a view that is read-write must also provide a reverse mapping. Although it may be possible to use the concept of a view in supporting model compliance verification, in the context of this dissertation, we do not make use of views and therefore do not consider them any further⁸.

Figure 4 Entity-Relationship view of the Bank Account model (Figure 2)

⁸ This does not imply that views could not be used for model compliance verification, rather that they are not used in our approach.

Transformations

A transformation takes one or more input models and manipulates them to either update the input models or produce a new model. The QVT Merge submission defines two sub-types of the transformation concept: relation and mapping.

Relation

A relation is a multi-directional transformation specification, which does not generate or modify a model. That is, a relation is not executable in the sense that it does not produce a new model or modify one of the input models. Instead it produces a success or failure result. Relations are able to evaluate consistency between two or more models [19, 31]. Therefore, relations are useful for validating and/or verifying consistency or compliance between a set of models. That is, one can define a set of relations that will evaluate whether a set of conditions holds between two or more input models.

For example, suppose that we have UML *design* model and an *implementation* model specific to JAVA (i.e., the metamodel is the JAVA metamodel [15]). In order to maintain consistency between the two models we could want to ensure that the names used for classes and their properties in the *design* model are carried through to the names used in the *implementation* model. We can define a relation that takes as input the *design* model and the *implementation* model and iterates through each of the classes in the *design* model to make sure that they also appear in the *implementation* model.

Mapping

A mapping is a uni-directional transformation implementation that can create or manipulate a model [19]. That is a mapping is executable in the sense that it produces a new model or modifies one or more of the input models. A mapping may be a refinement of one or more relations, in which case it must be consistent with the relations that it refines. A mapping is consistent with a relation when its domains are a proper subset of the domains defined in the

relation and if and only if after executing the mapping the product satisfies the relation that it refines.

Mappings are useful for refining a model to provide a greater level of detail. They can also be used to transform a model represented in a particular modeling in language(s) into a model represented in a different modeling language. For example, suppose that we have model that has been developed using the UML and the chosen implementation language is the RoseRT language. It is possible to create a set of mappings that can transform the UML model into the RoseRT language, retaining all of the properties and elements that are desired by the mapping writer. The advantage of defining mappings in such a situation is reuse; if we were to transform the UML model by hand the next time it would also have to be done by hand. By writing mappings that can be automated, the process is repeatable.

2.4.1.3 *QVT as Standard Transformation Language*

The QVT Merge submission defines a standard language for defining transformations (relations and mappings). The submission hereafter referred to as QVT makes use of pattern matching in defining both relations and mappings. The language provides both a textual and graphical form. The graphical form can be considered an abstraction of the textual form.

The QVT pattern expression language provides a powerful mechanism for selecting particular elements from the input models. That is, a pattern expression defines at the metamodel level the conditions under which model elements are matched. The matched elements are then used by the mapping or relation in which the pattern expression is defined.

In order to increase the usability and expressivity of the pattern expression language in QVT the pattern expressions defined in a transformation, known as domains, are accompanied by a condition that must be satisfied. Thus, a transformation is defined by a set of domains (pattern expressions) and a single condition that covers all domains. The condition can also be used to relate the domains to one another. Therefore, a relation is only successful and mapping is only executed when all domains and the condition are satisfied. In Figure 5, a mapping definition is

shown with the domain and condition expression explained. In order for this mapping to be executed the parameter 'class', which is passed into the mapping must contain the fields *name*, *attributes* and *associations* each of which is bound to the free variable *n*, *a* and *assoc* respectively. The value assigned to the *name* field is bound to the free variable *n*, and the condition expression states that *n* must be equal to the string "Account".

Figure 5 Mapping explaining the pattern expression and condition expression

The free variables bindings are created in the domain expressions can be used in other parts of the mapping or relation definition. For example, in the previous paragraph the free variable *n* was used in the condition expression. A mapping or relation can have a free variable that appears in more than one domain, for example Figure 6. In such a situation the values of the two must unify otherwise the pattern expressions are not satisfied. Consider again our example in Figure 6 the name field of both the first domain and the second domain must unify in order for the relation to be satisfied. If they unify the value is bound to *n*.

Figure 6 Example of a free variable that appears in two domains

The most basic difference between a relation and a mapping in QVT is that a mapping also includes a body. The body of a mapping defines how to produce the resulting model or model element, or it defines how an input model is to be manipulated. A mapping is defined through a mix of imperative (body of mapping) and declarative (domains and conditions) statements. The definition of a relation leans more towards a pure declarative approach (domains and conditions). We now define in more detail relations and mappings.

Relation

Figure 7 Syntax of a QVT relation definition

A relation definition is defined by a mandatory name that must be unique within its evaluation context. The number of domains in the relation's definition implicitly specifies the number of parameters for a relation. Therefore, if a relation definition contains three domains then it expects three inputs. The domains are patterns that along with the optional guard condition ("when expression") define the result of a relation evaluation. More specifically, the domains define the expected type and fields of the input values. The fields are bound to free variables, which must unify if a free variable appears in more than one domain. If unification is not possible then the relation returns a failure result.

A relation definition can also contain an optional "when expression". The "when expression" is a Boolean condition that relates the domains. If the "when expression" evaluates to true, then a success result is returned from the relation; otherwise a failure result is returned. If no "when expression" is specified within the relation definition and the domains unify, then a success result is returned by the relation. The "when expression" follows closely the syntax of OCL expressions and therefore we do not discuss it here.

We note that it is possible that a set of values be passed into a relation. When this happens each member of the set is considered as a candidate for satisfying the relation. That is, each member of the set will be used as the input to the relation evaluation until either the relation is successful or there are no more elements in the set, in which case the relation returns a failure result. If more than one parameter is a set, all possible combinations⁹ will be attempted before returning a failure result!

---

⁹ The QVT Merge submission does not address the potential for combinatorial explosion that this creates.

Mapping

```

mapping <mapping_name>(<parameter> (, <parameter>)*)
{
    domain ( <object pattern> } )
    ( domain { <object pattern> } ) *
    ( where { <expression> } ) ?
    body { <expression> }
}

```

Figure 8 Syntax of a QVT mapping definition

A mapping definition is defined by a mandatory name that must be unique within its execution context. The definition also contains a non-empty list of untyped input parameters that define the number of inputs expected when the mapping is invoked. More specifically, the list of parameters defines the names that can be used within the mapping definition to reference an input value. Within the definition of the mapping is a non-empty list of domains, which are patterns that define the conditions along with the optional guard condition (“where expression”) under which the mapping is invoked. In order for the mapping to be invoked and produce a result, all of the domains must be satisfied and, if it exists, the “where expression” must evaluate to true, otherwise the mapping will not execute. Recall that the domains create free variable bindings that must unify, and if unification is not possible then the domains are not satisfied.

The number of domains must be equal to or less than the number of parameters. And the first parameter is matched against the first domain, the second parameter against the second domain and so on. The additional parameters are considered to always match (i.e., their domain would always be satisfied and therefore is not included).

The “where expression” is a Boolean expression that constrains the invocation of the mapping. The expression can contain references to the parameters, as well as to the free variables bound in the domain evaluations. The expression follows closely the syntax of OCL expressions and therefore we do not discuss it here.

The body expression defines how the output is created. That is, the evaluation of the “body expression” produces the result of the mapping. The result of evaluating the “body expression” can be an integer, a string, a Boolean, an object or a set. Once again the syntax of the “body expression” follows the OCL expression syntax. The one exception is an object expression (e.g. Figure 9) whose evaluation results in an object. An object expression defines the type of the object inside of the parentheses and then the fields of the object are defined within brackets. Evaluating the expression on the left hand side of the equal sign produces the value assigned to the field. The expression defining a field can be the invocation of another mapping; Figure 9 contains an example of this.

Figure 9 QVT mapping definition example with an object expression in the body

2.4.1.4 *QVT Further Reading*

For the reader who is interested; more information on the QVT can be found in:

- [36] Object Management Group (OMG), "MOF 2.0 Query / Views / Transformations RFP,"
- [19] Tracy Gardner and Catherine Griffin, "A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard,"
- [31] Adaptive Ltd., Alcatel, CBOP, Codagen Technologies Corporation, DSTC, International Business Machines, Softbeam, TCS America, TNI-Valiosys, and Thales, "MOF Query / Views / Transformations First Revised Submission,"
- [46] QVT Partners Website

2.4.2 Object Constraint Language

The Object Constraint Language (OCL) 2.0 [39] is a constraint specification language with precise semantics [68]. More specifically, OCL expressions evaluate without side effects. That is, the state of a system can never change due to the evaluation of an OCL expression, even though an OCL constraint can be used to specify a required state change. The OCL syntax and semantics is used extensively throughout the QVT and therefore we feel it is important to provide an overview in this section.

The OCL is not a programming language. It is not possible to write logic or control flow statements in OCL. A process or thread cannot be created and only query operations may be called. A query operation is an operation that does not have any side effects. As the OCL is a modeling language by definition, its expressions are not directly executable.

The OCL is a strongly typed language. Each OCL expression has a type. To be well formed, each OCL expression must conform to the OCL type rules [39]:

- A type, *type1* conforms to the type, *type2* when an instance of *type1* can be substituted at each place where an instance of *type2* is expected;
- Each type conforms to each of its supertypes; and
- Type conformance is transitive: if *type1* conforms to *type2*, and *type2* conforms to *type3*, then *type1* conforms to *type3*.

The second version of the OCL was standardized by the OMG in August of 2003. Its main use is for adding constraints to UML models [38, 40]. Additionally, the OCL can be used to define derived attributes and query operations.

More specifically the OCL can be used for a number of purposes including:

- To specify invariants on classes and types in a class model;
- To describe the preconditions and postconditions on operations;
- To specify query operations;
- To specify derivation rules for attributes;
- To specify type invariants for stereotypes;
- To describe guards (i.e. conditions on transitions in a statechart); and
- To specify target (sets) for messages and actions.

In this dissertation, we focus on the expression syntax of the OCL language as it is the basis for expressions in the QVT. The expression syntax provides mechanisms for working with collections and iterators, standard mathematical and logical operators, and for referencing and navigating model elements among several others. The three tables that follow (Table 2, Table 3

and Table 4) briefly explain some of the expression elements that are relevant to this dissertation, for a more complete discussion of the OCL expression syntax and semantics including examples see [39, 68].

Table 2 OCL Operators that are relevant to this dissertation

Operator	Explanation
or	The standard logical or operator whose resulting type is a Boolean value.
and	The standard logical and operator whose resulting type is a Boolean value.
not	The standard logical negation operation whose resulting type is a Boolean value.
=	The equality operator that checks for equality between its left hand operand and right hand operand. Its resulting type is a Boolean value.
!=	The not equal operator that checks that the left and right operands are not equal. Its resulting type is a Boolean value.
+	The addition operator. When the left and right operands are Integers then the result is the Integer value that results from adding the two operands together. If the left and right operands are Strings then the result is a new String that results from appending the right operand to the end of the left operand.
.	The navigation operator. The left operand is an Object and the right operand is the attribute or operation to access in the object. Its return type is the type of the field or operation that is accessed.
->	The operation invocation or iterator for a set. Therefore, the left operand is a set and the right operand is either a set operation or iterator. The return type is determined by the operation or iterator.

Table 3 OCL collection operations that are relevant to this dissertation

Operations	Explanation
isEmpty()	Returns a true result if the collection contains no elements and false otherwise.
notEmpty()	Returns a true result if the collection contains at least one element and false otherwise.
size()	Returns an Integer value that is the number of elements in the set.
includes(<object>)	Returns a Boolean value indicating whether the <object> is a member of the collection.
excludes(<object>)	Returns a Boolean value indicating whether the <object> is not a member of the collection.
at(<int>)	Returns the Object at the <int> index.
first()	Returns the first Object.
last()	Returns the last Object.
flatten()	Returns a new set that results from flattening the source collection.
including(<object>)	Returns a new set that results from adding <object> to the source collection.

Table 4 OCL collection iterators that are relevant to this dissertation

Iterator	Explanation
forAll	Iterates over the elements in the collection and returns a true value if they all satisfy a given condition and false if any of the elements does not satisfy the condition.
select	Iterates over the elements and creates a new collection that contains those elements from the source collection that satisfy a given condition.
collect	Iterates over the elements and creates a new collection that contains the results of evaluating an expression for each element.
iterate	Iterates over the elements in the source collection and evaluates an expression.

2.4.2.1 OCL Further Reading

For the reader who is interested; more information on the OCL can be found in:

- [39] Object Management Group (OMG), "UML 2.0 OCL Specification, Final Adopted Specification,"
- [68] Jos Warner and Anneke Kleppe, The Object Constraint Language : Getting Your Models Ready for MDA
- [8] Klasse Objecten : OCL Center, <http://www.klasse.nl/ocl/>, 2004.

2.4.3 Unified Modeling Language (UML)

In this section we review the subset of the Unified Modeling Language (UML) that is most relevant to this dissertation. In particular we discuss the UML 1.5 modeling elements that are relevant to the case study presented in Chapter 4. The elements used in the case study, with the exception of sequence diagrams, are influenced by an examination of several industrial specifications found in [70].

The UML [38, 40, 45] is a modeling language standardized by the OMG and was intended for modeling software systems. It has since become the defacto standard for modeling software systems, with substantial tool support from vendors and academic attention with conferences dedicated to UML. The current UML standard is version 1.5 [45], however, the language has

received a major overhaul to expand its scope of use. This overhaul will result in the UML 2.0 [38, 40], which is currently in its finalization phase.

The UML provides graphical notations for capturing many aspects of a software system. One such aspect is system structure, which is captured through *Class* diagrams, *Object* diagrams, *Package* diagrams, *Component* diagrams, and the new *Composite Structure* diagrams. A second aspect is system behaviour, which is modeled with *Activity* diagrams, *Use Case* diagrams, *Interaction (Sequence, Collaboration, Interaction Overview and Timing)* diagrams and *State Machine* diagrams.

The Class diagram model elements and Sequence diagram¹⁰ model elements are the most relevant to this dissertation, therefore will restrict our discussion to these elements. A class model allows a software engineer to capture a certain aspects of the system structure. Most notably the class diagram captures the classes and interfaces, their properties (attributes and operations) and the relationships that exist between classes and interfaces.

The attributes of a class correspond to its properties, either specific to one instance or shared by all instances of the class. A name, a type and its visibility to other entities in the system define each of a class's attributes. There are other properties of attributes such as multiplicity and changeability; however, they are not relevant to this dissertation. We refer the interested reader to [40, 45].

The operations of a class correspond to its capabilities, again either specific to one instance or shared by all instances of the class. An operation is defined by its name and a set of parameters. This set of parameters contains a name, type and kind for each parameter defining the operation. A parameter can be an input parameter, an output parameter or both. Additionally the set of parameters may contain a return parameter, which defines the operations return type. There are other properties of attributes such as *isQuery* and *isAbstract*, however, they are not relevant to this dissertation. We refer the interested reader to [40, 45].

---

¹⁰ Sequence diagrams contain information that is useful in transforming a UML 1.5 specification into a RoseRT specification and appear in some specifications (for example the SCA).

A class model also contains several types of relationships between two or more classes in the model: associations and generalizations. An association defines a semantic relationship between two or more classes. Each end of an association has a multiplicity specifying the number of instances at that end that are associated with the Class at the opposite end. Each end also has a name that indicates the role that it plays with respect to the instances at the opposite end of the association. One end of an association relationship can be labelled as an aggregate (instances can exist on their own) or composite (instances cannot be shared by more than one instance at the opposing end).

The generalization relationship is used to capture that one class is more general form of a second class. That is, the 'parent' in the generalization relationship is more general than the 'child', which is a specialization of the parent. Typically, this also indicates that an instance of the more specific class can be used in place of or wherever the more general class is allowed.

Although we have focused the previous discussion on classes, interfaces can also have attributes and operations. Interfaces and classes can also be related to one another through association relationships. However, a class interface can specialize an interface, but an interface cannot specialize a class. A Class that specializes an Interface is said to realize the Interface.

A sequence diagram illustrates how objects interact with one another. Its primary focus is on the exchange of messages between objects and more importantly sequences of messages between objects. Two axes define a sequence diagram: the horizontal axis defines the set of objects in the interaction and the vertical axis shows the sequence of messages exchanged between the objects on the horizontal axis. That is a sequence diagram illustrates an interaction pattern for a specific scenario that happens at some point in time in the system's execution.

Each message in a sequence diagram is defined by two events: send event and receive event. The object that is the source of the message triggers the send event and the received event occurs when the receiving object gets the message. The send event must occur before the receive event. Additionally, a message can be linked to an operation defined by the receiving object. It is messages linked to operations that we are interested in this dissertation. A message is linked to an

operation when the message references an operation in the receiving object. Figure 10 shows an example of a sequence diagram.

Figure 10 Sequence diagram example

This overview of UML has been brief as we assume the reader has some knowledge of the UML. More detailed discussion of some elements of the UML is provided throughout this dissertation as required.

2.4.3.1 UML Further Reading

For the reader who is interested; more information on the UML can be found in:

- [38] Object Management Group (OMG), "UML 2.0 Infrastructure Final Adopted Specification,"
- [40] Object Management Group (OMG), "UML 2.0 Superstructure Final Adopted Specification,"
- [45] Object Management Group (OMG), "Unified Modeling Language (UML), Version 1.5,"

- [50] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, 2nd ed
- [16] Hans-Erik Eriksson, *UML 2 Toolkit*

2.4.4 IBM Rational Rose Real-Time

The IBM Rational Rose Real-Time toolset [56]; hereafter referred to as RoseRT; provides the concepts for modeling complex reactive systems. One of its distinguishing characteristics is model executability. That is, the software engineer is able to generate a running system from the models they have defined in the tool. The modeling language provided in the tool closely resembles the UML modeling language, even more so with the addition of Composite Structure diagrams in UML 2.0. However, to the best of our knowledge no explicit metamodel exists in the public domain. Therefore, for the purposes of this dissertation a metamodel has been extracted from the tools documentation. We also assume that the reader may be less familiar with some aspects of the RoseRT modeling language and therefore we provide a more in depth overview.

A RoseRT model consists of a structure modeled by Packages, Capsules, Classes and Protocols. The model also contains a behavioural definition for each Capsule, which is modeled through a state machine. In this dissertation we are concerned with the structural elements of a RoseRT model and thus do not discuss any further the behavioural elements.

Classes and their Associations are closely related to Classes and Associations in the UML, which are discussed in Section 2.4.3 and therefore we focus our attention on Capsules, Capsule Roles, Ports and Protocols.

2.4.4.1 Capsules

Capsules represent the central modeling construct of RoseRT modeling. They are the active objects in the system and are defined as independent, concurrently active logical machines. A capsule is similar to a class, however, a capsule only has private and protected attributes and

operations. Communication between a capsule and its environment is achieved through synchronous and asynchronous message passing. Messages are sent and received on public and protected ports; these ports ensure two-way encapsulation.

A capsule has two views: an internal view and an external view. The external view (e.g. Figure 11) describes the public interface of the capsule and encapsulates the internal structure and behaviour of the capsule. It consists of a set of public ports (e.g. a and b) on, which messages can be sent *in* to the capsule and come *out* of the capsule. It does not describe how the capsule handles or sends the messages.

Figure 11 Capsule A's external view

The internal view (e.g. Figure 12) gives the structural decomposition of the capsule as well as its behaviour. The behaviour is described by means of a state machine. The structural decomposition consists of a set of ports (e.g. a, b and c), a set of contained capsules or capsule roles (e.g. B and C) and a set of connectors (e.g. c1, c2, c3). The structural decomposition is recursive, meaning the decomposition can be as deep as needed. Alternatively, the internal and external view can be seen in a class diagram.

Figure 12 Capsule A's internal view

A capsule's set of ports can be sub-divided into a set of public end ports, a set of public relay ports and set of protected end ports. The public end ports (e.g. a) and the protected end ports (e.g. c) are the interface to the capsule's behaviour. The public relay ports (e.g. b) forward received messages to the connected contained capsule. The protected end ports allow a capsule to have control over certain elements (contained capsule's) of its internal structure.

The contained capsules (known as capsule roles) play specific roles in their container capsule. A fixed capsule role is created when the capsule is created and destroyed when the capsule is destroyed. There are two other types of capsule roles: optional and plug-in. An optional capsule role requires the explicit creation by its container capsule and may be destroyed explicitly or when its container is destroyed. A plug-in capsule role acts as a placeholder that can be filled by a capsule created outside the container. The plug-in roles concept allows for multiple containment¹¹ and dynamic structures.

A capsule's connectors model the communication that exists between two capsule roles. It connects two capsule roles via their public ports. A connector may also be used to connect a relay

¹¹ Multiple containment allows a capsule instance to play a role in multiple container capsules.

port with a capsule role. In order for two capsule roles or relay port and capsule role to be connected, the connected ports must be compatible. Compatibility of ports is considered elsewhere in this section on RoseRT (Section 2.4.4.5 on Connectors).

A traditional class may have a set of attributes and operations associated with it. These may be either public, protected, private, etc. A capsule may have a set of attributes and operations with the restriction that they are all private or protected. This is a necessity for total encapsulation.

2.4.4.2 *Capsule Roles*

A capsule role is a placeholder for an instance of a capsule to play a specific role for another capsule, its container. It has a name that should identify the role that it is playing, for example in Figure 12; we have two capsule roles with names C and B. As we mentioned in the previous section a capsule role is either fixed, optional or plug-in.

A container capsule may require more than one instance of a capsule to play a role. This can be modelled by setting the cardinality of a capsule role to the desired number. The cardinality of a capsule role is finite and defined at compile time.

2.4.4.3 *Protocols*

A RoseRT protocol class is a specification for all capsules that offer a common public interface. It is defined as a set of incoming signals and a set of outgoing signals. A signal has a name and an optional parameter (limit of one). Figure 13 is an example of a protocol. Its name is ProtA, its set of incoming signals is (e1, e4) and set of outgoing signals is (e2, e3). Signal e4 has a parameter of type char and signal e2 a parameter of type int.

Figure 13 Protocol example

2.4.4.4 Ports

In the RoseRT a port denotes a capsule's interface. A port constitutes the means by which a capsule can communicate with its environment, and with its internal structure. A port implements a protocol and defines the sole means of communication with a capsule. Ports encapsulate internal behaviour leaving an external observer to deduce a capsule's behaviour by the flow of signals on its ports. Conversely, a capsule's perception of its surroundings is limited to the information received on its ports.

Two types of ports exist: end ports and relay ports. End ports are ports on capsules that are the source or destination of signals. An end port can either be public (externally visible) or protected (internally visible). The relay port, involves the use of an intermediate interface to relay a signal to a capsule role. Relay ports are used to delegate the handling of all signals received on that port to a capsule role. End ports are illustrated using a square with a small state attached, while relay ports are illustrated with a square.

A protocol class defines the incoming and outgoing signals of a port. It may be desirable for a capsule to have the incoming messages of a protocol as the outgoing signals and the outgoing the incoming. This is supported through the notion of base and conjugated ports. A base port implements the protocol as is, while a conjugated port swaps the incoming and outgoing signals.

2.4.4.5 Connectors

A connector is used to connect two compatible ports. Two ports are compatible if the protocols they implement are compatible. Two protocols, A and B, are compatible if A's set of outgoing signals is a subset of B's incoming signals and B's set of outgoing signals is a subset of A's incoming signals. This ensures that all the messages sent on one end of the connector can be received at the other end of the connector. A connector offers no notion of communication direction; it may be bi-directional or uni-directional depending on the protocols they implement.

Direct communication between two capsule roles in a container may occur only if a connection between them exists. Indirect communication may occur through the container capsule.

2.4.4.6 RoseRT Further Reading

For the reader who is interested; more information on RoseRT can be found in:

- [53] Bran Selic, Garth Gullekson, and Paul T. Ward, *Real-time Object-Oriented Modeling*
- [56] IBM Rational Software Rational Rose Real-Time Documentation
- [54] Bran Selic and James Rumbaugh, *Using UML for Modeling Complex Real-Time Systems*

2.5 A Combinatorial Test Pattern

One aspect of our model compliance verification approach is verifying that a mapping or compliance rule produces the correct result. In this section we discuss a pattern for building tests that we will use as a base for our approach to verifying mappings and relations.

The Category-Partition Test Pattern Chapter 10 of [2], defines a pattern for designing method scope tests. The tests are based on the inputs and outputs of the Method Under Test (MUT).

More specifically each input parameter is analysed to identify categories for the parameter. A category is defined to be a subset of the parameter's possible values and does not overlap with any other category. Each category is then partitioned into a set of choices, which are specific test values. For example, a set may have a *boundary cases* category, whose choices are *empty*, *singleton* and *full*. The actual choices are derived from domain knowledge and experience, which is outside the scope of this dissertation.

Once the choices for each parameter have been defined, constraints and dependencies between the choices are identified. Using the choices for each parameter and the constraints and dependencies that have been identified, a combinatorial test model is built. The combinatorial model contains a row for each valid combination of choices. Invalid choices are identified by the constraints and dependencies. Often dependencies cause the value of a particular parameter to be inconsequential to the particular outcome in which case we don't care about the choice for that parameter. This is modeled by a DC (for don't care) for that parameter in that particular row.

Once the input rows have been created in the combinatorial model the expected results must be defined. That is, for each row in the table the expected result for the method given the input values in the row must be stated. This combinatorial model now provides a suite of tests that can be used to test that given a set of inputs the method produces the expected result. For a more detailed explanation and the theory (of equivalence partitioning) behind the pattern see Chapter 6 and Chapter 10 of [2]. This pattern will be useful for our approach to model compliance verification.

2.6 Model-To-Model Transformation Tools

To our knowledge no tools exist with similar capabilities to the prototype we propose in this dissertation. However, there are several tools or prototypes that are relevant. The most relevant type of tool to this dissertation is a model-to-model transformation tool. Several such tools exist, each with a different approach to model-to-model transformation. In this section we provide a brief overview for each of these tools capabilities.

In [47] the authors define a model manipulation scripting language that is influenced by prototyping languages like Python. The advantages of this approach are the scripting language is modeling tool independent and it is interpreted. Since the language is interpreted a software engineer can modify his/her scripts and re-execute them without recompiling source code. However, unlike QVT, this scripting language is not slated to become an industry standard for defining model-to-model transformations.

The authors of [24, 60, 61, 64] describe a tool that supports a rich and expressive model transformation language MITL¹² (abstract syntax) and BasicMTL (concrete syntax). The tool surrounding BasicMTL is an Eclipse plug-in that currently compiles the BasicMTL syntax down to JAVA source code. The JAVA source code can then be compiled and executed. The BasicMTL language is influenced by the functional programming paradigm and provides facilities for easily navigating MOF based models and can load both models and metamodels from XMI. The actual model(s) that a BasicMTL program operates on can either be specified at compile time or at run-time. The syntax and semantics of this language appear to be richer and more powerful than QVT, but once again it is not slated to become an industry standard.

In [58] the authors demonstrate that it is possible to transform XMI documents directly. The approach relies on the XSL Transformations (XSLT) [66] and the XML Path Language (XPath) [65] standards and tools to transform one XMI document into another XMI document or other model format. While this approach appears very simple, a mapping definition can become complex and hard to understand very quickly.

Probably, the most relevant tool to this dissertation is QVT prototype tool [46] developed by the QVT Merge submission group. This particular tool provides support for a subset of the graphical QVT notation and is a proof of concept for their submission. This means that many of the OCL operators and expressions are not supported, nor are the more advanced graphical syntax constructs.

---

¹² Model Transformation Language

Finally, several authors including [20, 69] define formal model transformation approaches based on logic programming. Most of these approaches require defining transformations using a very restrictive and mathematical syntax.

Chapter 3

TOOL SUPPORT FOR MODEL COMPLIANCE VERIFICATION: USING QVT

In this chapter, we describe the capabilities of our proposed prototype that supports the activity of model compliance verification. Initially the chapter presents an overview of our prototype, which is influenced by the model compliance verification approach described in Section 2.3. A comparison with an existing tool that supports model compliance is provided. The chapter concludes with a discussion of how the prototype tool can address specific challenges in model compliance verification that we have previously defined in [71]. Any changes whether major or minor to the process described in [70] and Section 2.3 due to proposed prototype are made explicit in this section.

3.1 A Tool to Support Model Compliance Verification

In this section, we describe in detail the capabilities of the prototype that we have built to support the activity of model compliance verification. We first provide an overview of our prototype's capabilities in general and this includes any artefacts that the prototype produces (for example the traceability model). This overview is followed by a description of how the tool supports each of the activities that make up model compliance verification.

3.1.1 A prototype tool for Model Compliance Verification

This section describes the prototype tool that we have built to support the activity of model compliance verification. We start by putting forth a use case for the development of the prototype.

A process for model compliance verification has been defined where a design model is verified for compliance with a specification model. It is highly likely that the two models use different modeling languages. A tool to support such a process must be able to handle models that use modeling languages with a well defined metamodel while at the same time provides a mapping and a compliance rule definition language that is accommodating to these differing modeling languages.

The prototype must be able to support the software engineer in each of the activities that are defined as part of the overall activity of model compliance verification. That is, not only must the prototype be able to handle the various modeling languages and definition of the mappings and compliance rules, it must also be able to apply the mappings to a specification model to produce a transformed specification model. The software engineer must also be able to execute the compliance rules using the transformed specification model and design model as inputs to produce an answer to the compliance question.

This use case defines several high-level objectives that guided the development of our prototype. The objectives are listed in this section and revisited after the prototype has been described. The high-level objectives include:

1. The ability to import, define, manipulate and save *specification*, *tSpec* and *design* models;
2. The ability to define and modify mappings that transform the *specification* into a semantically related *specification* model (*tSpec*) in the *design* modeling language;
3. The ability to define and modify compliance rules that define the semantics of compliance between a *design* model and a *tSpec* model;
4. The ability to execute the defined mappings on a *specification* model to produce the *tSpec* model; and
5. The ability to answer the question “Is my *design* model compliant with the *specification* model?” by evaluating the compliance rules using the *tSpec* and *design* models as input.

3.1.1.1 The Prototype

From a high-level point of view the prototype consists of three major components: a Model Compliance Verification Integrated Development Environment (IDE), a set of model converters and a QVT engine. Each of these components serves a specific purpose in realizing the tool's objectives. The diagram in Figure 14 provides an overview of the relationship between the components. The Compliance Verification IDE is the interface with which the software engineer interacts. It depends on a set of Model Converters; each member can extract a model from a specific modeling tool. For the purposes of this dissertation two model converters were created and are shown in the diagram. Additionally the IDE depends on the QVT engine, which is able to execute and evaluate the mappings and compliance rules respectively.

Figure 14 Overview of the tools individual components

Compliance Verification IDE

As previously stated the Compliance Verification IDE is the component in the prototype with which a software engineer interacts. It provides a Graphical User Interface (GUI) component; a screen capture is shown in Figure 83 found in Appendix B. The IDE's core concept is a Model

Compliance Verification project, which can be created, saved and loaded through the IDE. Additionally, the IDE provides the ability to create, extract, manipulate and save models. The diagram in Figure 15 shows the three components that are contained within the IDE component: GUI, Model Inspector and Project Explorer.

Figure 15 Overview of the compliance verification IDE components

The Project Explorer component facilitates the management of Model Compliance Verification projects. It provides the software engineer the ability to import, create and save models and to create, set and add the mappings and compliance rules to a project. That is, the Project Explorer manages the data necessary for performing the activity of Model Compliance Verification.

A Compliance Verification project defines the models, mappings and compliance rules that are involved in the verification of compliance for a particular *specification* model and *design* model. As shown in Figure 16 a project consists of a set of models, a set of mappings and a set of compliance rules. The software engineer is given the ability to specify the *specification* model, *tSpec* model, *design* model and an *additional information* model. All of the four models can be extracted from another tool through the appropriate model converter, loaded from a previously created model or created within the tool. Once imported or created the model can be saved and loaded to/from an XML file. The *specification* model, *tSpec* model and *design* model are as described in the model compliance verification process (Section 2.3).

The *additional information* model provides the software engineer with the ability to specify additional information that can be incorporated into the *specification* to *tSpec* transformation process. This *additional information* model is not restricted to any particular metamodel or modeling language; it is left to the software engineer to define the metamodel and the information stored within the model. For example, the specification language and design language may use different names for equivalent primitive types. The *specification* model to *tSpec* model transformation needs additional information that states how to rename such types, for more information see the Type Correspondence challenge in Section 3.3.2.

Figure 16 The elements of a Model Compliance project

The mappings component of a project defines the set of mappings used to transform the *specification* model into the *tSpec* model. The mappings are defined by a single mapping that starts the transformation process (*Mapping starting point*) and a set of support mappings (*Mappings*). The *Mapping starting point*, which fires the mapping process is like the main function in C++; the entry point for program execution. Similarly, the compliance rules component is a set of rules that define the semantics of compliance between a *design* model and a *tSpec* model. The compliance rules also consist of a single rule that starts the compliance analysis (*Compliance starting point*) and a set of supporting rules (*Compliance rules*).

For each of the four models a Model Inspector is provided. The Model Inspector component provides the ability to manipulate and navigate models. That is the software engineer has the ability to add, modify and delete elements from a model through a Model Inspector. When a model is set within the Project Explorer it is loaded into the appropriate Model Inspector. Changes made to a model that was imported from an external tool through a Model Converter are not reflected in that tool's model.

Within a Model Inspector the model is represented as a set of objects each containing a set of slots. A slot is given a name and contains a value. A value is a primitive value (Integer, String or Boolean), an object, an object reference or a set of values. This representation allows the software engineer to represent models from most modeling languages. That is, the Model Inspector does not restrict the software engineer to any particular metamodel¹³. The tree structure that the Model Inspector uses is not ideal for a software engineer to navigate or manipulate a model. However, this limitation does not change the results of our research.

Model Converter

The Model Converter components act as bridges between the proposed prototype and external modeling tools. That is, a software engineer that wishes to import a model from a particular modeling tool has only to write a bridge (Model Converter) that the prototype invokes to extract the model. For the purposes of this dissertation two model converters have been written: one for IBM Rational Rose and another for IBM Rational Rose Real-Time.

Each model converter shares a common interface for extracting the model. The interface specifies two operations: import and write. The import operation is mandatory and when invoked for a particular model will return the model in the prototype's internal model representation. That is, a model converter converts a model into objects and slots that contain values. For example, when the model pictured in Figure 17 is loaded into the prototype it produces a model inspector similar to the one presented in Figure 18.

---

¹³ The QVT requires that the metamodel be MOF compliant.

Figure 17 A bank system model using a simplified UML language

The write operation if implemented will create a model for the particular tool from the prototype's internal model representation (objects and slots). That is, the write operation provides the ability to create a model for the specific tool from a model within the prototype. Within the context of this dissertation the IBM Rational Rose Real-Time model converter provides the ability to write models to the IBM Rational Rose Real-Time toolset.

To summarize, a model converter is a component that converts a model from a particular modeling tools representation of a model into the prototype's internal representation of a model. The prototype tool represents a model in the form of objects and slots that contain values. A model converter may also provide the ability to create a model for the particular tool from the prototype's internal representation. For example, it is possible to create model converters for XMI [43] or Eclipse Modeling Framework (EMF) [5] models so that those models can be used within the prototype tool. In order for the prototype to recognize a new model converter it must be recompiled.¹⁴ This is undesirable and therefore we suggest here an alternative.

¹⁴ The prototype would require modifications to the context menus in the project explorer. That is, plug-in architecture has not been implemented at this point.

Figure 18 Screen capture of the model inspector after loading the BankSystem (Figure 17)

In Appendix D we present the XML schema for the prototype's saved models. Therefore, in order to integrate additional modeling tools into the current prototype a software engineer can develop a model converter that produces an XML model file that can be loaded by the prototype. That is, instead of the prototype invoking the model converter, the model converter would produce an XML file that is compliant with the schema in Appendix D. The prototype can then load the file as a model. We consider these to be external model converters.

QVT Engine

Section 2.4.1 provided an overview of the motivation for the Query, Views and Transformations (QVT) standard. Recall that one of the goals of QVT is to define the syntax and semantics of a

language or set of languages for defining model mappings and model relations. A model mapping is a set of instructions for creating a target model from a non-empty set of input models or for manipulating a non-empty set of input models. A relation defines a set of rules that associate the elements of two or more models. The result of evaluating a model relation is either a success or a fail, that is, either the models satisfy all of the relation conditions or they do not.

Within our prototype tool the QVT language for mappings and relations serves as the definition language for both mappings and compliance rules. It was chosen for several reasons; the main reason is that it will eventually become the de facto language for defining model transformations. Furthermore, it is ideal to reuse a standard language when it fulfills all of the relevant requirements¹⁵ rather than to create a prototype specific language.

As a consequence of using the QVT language as our mapping and compliance rule definition language, the prototype requires an interpreter that can execute and evaluate the mappings and compliance rules. The prototype has a QVT engine component that acts as a QVT interpreter; it understands mappings and relations written in the QVT language and can execute and evaluate them. We follow the evaluation semantics for QVT defined in the April 2004 submission [31] by the QVT Merge group.

The QVT engine takes as input a set of relations or mappings (transformations) one of which is the starting point. Each transformation exists in its own flat file; the interpreter resolves links or dependencies between the transformations. That is, a transformation that makes use of another transformation by invoking it is not required to specify it as an import; the interpreter resolves these imports. As a consequence, the name of each transformation in a project must be unique. It then evaluates the starting transformation against the model(s) passed in as input to the interpreter. The result of evaluating a mapping on the input model(s) is a single target model. Conversely, evaluating a relation on the input models is a Boolean result; if true the input models successfully satisfy the relation and false if the input models fail to satisfy the relation.

---

¹⁵ The syntax and semantics of the QVT language fits nicely with our criteria for a mapping and compliance rule definition language.

The execution of a mapping also produces a separate traceability [12, 30] model, this separation is the approach recommended in [13]. We feel that this approach is desirable since the two products (mapping execution result and traceability model) deal with separate aspects of the mapping execution. Furthermore, the target modeling language may not have the necessary concepts to embed the traceability information in the target model.

The traceability model captures the mapping that was applied, as well as, the elements used from the source model to produce a particular element in the target model. The model is a set of traceability items, where each item has the general form shown in Figure 19. The *source-objects* element of a *trace* is a non-empty set of model elements used by the mapping to create the target element. On the contrary the *target-objects* element of a trace is a possibly empty set of elements that were produced by the mapping. Each of the elements of the source and target objects sets specifies the type of element (*type*), the elements unique identifier (*valid*) and, if available, the name of the element (*name*). Figure 20 provides an example of a traceability model element; the mapping *OperationResultToSignal* was applied to the *confirm* Operation to produce the *confirm_RESULT* Signal.

```
<trace>
  <mapping-applied>MAPPING-NAME</mapping-applied>
  <source-objects>
    <obj type="VAL TYPE" valid="VALUE ID" name="VAL NAME" />
    ...
  </source-objects>
  <target-objects>
    <obj type="VAL TYPE" valid="VALUE ID" name="VAL NAME" />
    ...
  </target-objects>
</trace>
```

Figure 19 The general format of traceability items

```

<trace>
  <mapping-applied>OperationResultToSignal</mapping-applied>
  <source-objects>
    <obj type="Operation" valid="b0055283" name="confirm" />
  </source-objects>
  <target-objects>
    <obj type="Signal" valid="b48cb671" name="confirmResult" />
  </target-objects>
</trace>

```

Figure 20 Example of a traceability model element

When the QVT engine executes a mapping it produces a set of statistics. The set of statistics lists each of the mappings that were included in the evaluation (i.e. the starting point and the supporting mappings). For each mapping the number of successful evaluations (*success*) are indicated as well as the number of unsuccessful evaluations (*failure*). Figure 21 presents the general form of the mapping statistics produced by the QVT Engine and Figure 22 provides an example of the statistics produced by executing the UMLToRoseRT mapping on a source model. We can see that none of the mappings failed and for example that ClassToDataClass was executed 21 times successfully.

```

<mapping-stats>
  <mapping name="MAPPING NAME" success="# success" failure="# unsuccessful" />
  ...
</mapping-stats>

```

Figure 21 The general form of mapping statistics produced by QVT engine

```

<mapping-stats>
  <mapping name="UMLToRoseRT" success="1" failure="0" />
  <mapping name="PackageToLogicalPackage" success="2" failure="0" />
  <mapping name="ClassToDataClass" success="21" failure="0" />
  <mapping name="AttributeToAttribute" success="75" failure="0" />
  ...
</mapping-stats>

```

Figure 22 Example, a subset of the mapping stats produced for the evaluation of UMLToRoseRT on a model

If the evaluation of a relation fails the QVT engine produces a relation failure traceability model. This failure traceability model indicates the element in the first input model of the relation that could not be matched with an element in the other input models¹⁶. The model contains a set of trace items, which have the general form shown in Figure 23. Table 5 defines each of the attributes and elements of a relation failure traceability item. Figure 24 provides an example of a failure traceability model; the AttToAtt relation failed causing the ClassToClass relation to fail causing the PackageToNamespace relation to fail.

```

<trace-item relation=<RELATION NAME>
  e-name=<ELEMENT NAME>
  e-type=<ELEMENT TYPE>
  e-id=<ELEMENT ID>
  result=<RESULT>>
    <children />
</trace-item>

```

Figure 23 Relation failure traceability item general form

¹⁶ This makes the relation failure traceability model specific to our model compliance verification approach.

Table 5 Definition of relation evaluation failure traceability model elements

RELATION NAME	The name of the relation that failed.
ELEMENT NAME	The name of the element in the first input model that could not be matched in the second input model.
ELEMENT TYPE	The type of the element in the first input model that could not be matched in the second input model.
ELEMENT ID	The unique identifier of the element in the first input model that could not be matched in the second input model.
RESULT	The result of the relation evaluation should always be “FAIL” in this traceability model. This is because all successful items are suppressed from the model.
children	A trace item can have children; if it does it is the child element that caused the failure. This facilitates navigation within the input model to find the particular element that caused the failure. For example, in Figure 24 the root trace item states that the PackageToNamespace relation failed on the BankSystem:Package element, which was caused by the ClassToClass relation failing on the Account:Class element, caused by relation AttToAtt failing on the name:Attribute element.

```

<trace-item relation=PackageToNamespace e-name= BankSystem e-type= Package e-id=Pkg1 result=FAIL>
  <children>
    <trace-item relation= ClassToClass e-name= Account e-type= Class e-id=Cls1 result=FAIL>
      <children>
        <trace-item relation= AttToAtt e-name= name e-type= Attribute e-id=Att1 result=FAIL>
          <children />
        </children>
      </children>
    </children>
  </trace-item>

```

Figure 24 Example of a relation failure traceability model

3.1.2 Supporting Model Compliance Verification

In this section we describe in detail how the prototype tool that we have built supports each of the model compliance verification activities defined in [70]. Recall the activity of model compliance verification consists of:

1. Transforming a *specification* model into a semantically equivalent¹⁷ model (*tSpec*) in the *design* model's language;
2. Verifying that the *tSpec* is an accurate representation of the *specification* in the *design* model's language; and
3. Verifying that the *design* model is compliant with the *tSpec* model.

3.1.2.1 Transforming a specification model to a tSpec model

This activity involves transforming a *specification* model into a semantically equivalent¹⁷ model in the *design* language; the transformed *specification* is referred to as the *tSpec* model. This transformation bridges the semantic gap that may exist between the *specification* modeling language or metamodel and the *design* modeling language or metamodel. In [70], the author defines three activities that comprise this step: the definition of mapping rules, verifying that mappings are correctly applied and verifying the completeness of a set of mappings. We address the prototype's support for each of these activities individually.

Definition of Mapping Rules

The core of the prototype tool for supporting model compliance verification is the QVT engine (Section 3.1.1.1) that can execute QVT transformations (relations and mappings). Our support for defining mapping rules is through the QVT mapping definition language. A mapping is defined in a flat file, with one mapping per file. Recall that each project in the tool contains a set of support

---

¹⁷ Semantic equivalence must be defined and determined by a domain expert.

mappings (mapping files) and a mapping that is designated the starting point. The starting point is the mapping that begins the transformation process. The other mappings are used by the starting point in order generate the target model. The prototype provides a text editor for creating and editing mapping definitions.

When invoked the prototype executes the mapping starting point to create the *tSpec* model. For example, suppose the *specification* language is a simplified UML and that *design* language is a simple database modeling language that expresses the model in terms of tables, fields and relations. In order to produce the *tSpec* four mappings are necessary *UMLModelToDBModel*, *UMLClassToTable* (Figure 25), *UMLAttributeToField* and *UMLAssociationToRelation*. The *UMLModelToDBModel* mapping is the starting point that maps the UML model to a database model; Figure 25 presents an example of one of the mapping definitions. Within the tool the mapping starting point would be set to *UMLModelToDBModel* and the mappings set would include the other three mappings, as shown in Figure 26.

```

mapping UMLClassToTable(class)
{
  domain { (Class)[ name = n, attributes = a ] }
  body
  {
    (Table)
    {
      name = n,
      fields = a->collect(att | UMLAttributeToField(att))
    }
  }
}

```

Figure 25 UMLClassToTable.qvt mapping definition

Figure 26 Screen capture of mappings for UML to database model example

Correct application of mappings

Once a mapping is defined it is important to verify that when the mapping is applied to a particular input that the correct output is produced. This is similar to unit testing [21] class methods in Object-Oriented Programming (OOP) [22, 23]. Typically in OOP a unit test invokes a method on a class with a specific sets of parameter values and verifies the result of the invocation. In the context of model compliance verification we are concerned with unit testing the mappings that create a *tSpec* model from a *specification* model.

In OOP substantial research and development have gone into the development of unit testing tools. For example, the Java programming language [62] has the JUnit [28] tool that assists Java developers in unit testing the code they write. Similarly, for Microsoft's .NET platform [34] the NUnit tool [35] supports developers in unit testing code written in one of the languages supported by the .NET platform. Despite acknowledging that tools for unit testing in OOP exist currently our prototype forces the software engineer to unit test their mappings manually. In the remainder of this section we illustrate how a software engineer can manually verify his/her mappings within the tool.

Suppose a mapping, `UMLAttributeToCSharpAttribute`, is defined that transforms a UML attribute (`UMLAttribute`) into a C# attribute (`CSharpAttribute`). In order to ensure that the mapping is correctly applied, a combinatorial model [2] is built to ensure that the possible configurations of the input (`UMLAttribute`) produce their expected output. That is, assume our mapping is defined as shown Figure 28, which is based on the simplified metamodel presented in Figure 27. By analyzing the mapping and knowing the domain in which it is applied a combinatorial model is developed similar to the one shown in Table 6. The building of the combinatorial model follows the pattern presented in Section 2.5. The one exception is that categories are not used, only choices, and the choices are generic. For example, the choices for the name of an Attribute element are empty string and non-empty string, rather than “” (empty string) and “*accountBalance*”. We consider “*accountBalance*” to be an instantiation of a choice.

Figure 27 Example metamodels

Within the prototype a new project is defined specifically for the mapping under test (MUT). Using a combinatorial model developed to verify the correct application of the MUT, we model as the *specification* model in the prototype an object for each of the test cases (rows). This is considered to be an instantiation of the combinatorial model. Therefore, in the case of the model in Table 6 we would define ten (10) objects upon which the mapping will be invoked. This can either be done in a modeling tool, for example Rational Rose, or it can be done manually in the *specification* model inspector. A mapping that acts as a test driver is defined and becomes the *Mapping Starting point* for the project. For example, Figure 29 presents a mapping that acts as a driver for testing the mapping defined in Figure 28. The purpose of the driver is to iterate through the test objects, applying the MUT. A results model is produced when the driver is executed, the results model can be analyzed to verify the mapping was in fact applied correctly. Such an analysis is not automated and must be performed manually by the software engineer.

Ideally the analysis of the test results model would be automated. In order for such an analysis to be automated the expected result would have to be linked to the particular test case and provided in a machine-readable format. That is, the expected results for a particular MUT would have to be provided to our prototype in a format that it can interpret. Furthermore, the prototype would then compare the actual result model of executing the test case with the expected result. Supporting such automated analysis is left for future work.

```

mapping UMLAttributeToCSharpAttribute(attribute)
{
  domain { (UMLAttribute) [ name = n, type = t, visibility = v ] }
  body
  {
    if v = "public" or v = "protected" then
      Set{
        (CSharpAttribute) [name = n, type = t, visibility = "private"],
        (CSharpProperty) [name = n, type = t, visibility = v, modifiable = true]
      }
    else
      Set{
        (CSharpAttribute) [name = n, type = t, visibility = "private"]
      }
    endif
  }
}

```

Figure 28 Mapping to be verified for correct application

```

mapping UTest(attribute)
{
  domain {(UMLAttributeTest)[tests = t]}
  body
  {
    (TestResult)
    [
      results = t->collect(testCase |
        (TestCaseResult)
        [
          testcase = testCase.testcase,
          result =UMLAttributeToCSharpAttribute(testCase)
        ]
      )
    ]
  }
}

```

Figure 29 Driver for verifying mapping in Figure 28

Table 6 Mapping application verification combinatorial model, where each row represents a separate test case

	Input				Expected Result				
	Object Type	name	type	visibility	Object Type	name	type	visibility	modifiable
1	!UMLAttribute	D/C	D/C	D/C	FAIL				
2	UMLAttribute	Λ	Θ	+	CSharpAttribute	Λ	Θ	-	N/A
					CSharpProperty	Λ	Θ	+	true
3	UMLAttribute	" α "	Θ	+	CSharpAttribute	" α "	Θ	-	N/A
					CSharpProperty	" α "	Θ	+	true
4	UMLAttribute	Λ	" α "	+	CSharpAttribute	Λ	" α "	-	N/A
					CSharpProperty	Λ	" α "	+	true
5	UMLAttribute	Λ	Θ	#	CSharpAttribute	Λ	Θ	-	N/A
					CSharpProperty	Λ	Θ	#	true
6	UMLAttribute	" α "	Θ	#	CSharpAttribute	" α "	Θ	-	N/A
					CSharpProperty	" α "	Θ	#	true
7	UMLAttribute	Λ	" α "	#	CSharpAttribute	Λ	" α "	-	N/A
					CSharpProperty	Λ	" α "	#	true
8	UMLAttribute	Λ	Θ	-	CSharpAttribute	Λ	Θ	-	N/A
9	UMLAttribute	" α "	Θ	-	CSharpAttribute	" α "	Θ	-	N/A
10	UMLAttribute	Λ	" α "	-	CSharpAttribute	Λ	" α "	-	N/A

Table 7 Explanation of symbols used in Table 6

Notation	
Λ	any name value, but it is the same for a particular row
Θ	any type, but it is the same for a particular row
D/C	value is irrelevant to expected result, therefore can be any valid value for the type
N/A	Field is not applicable for this type of object
!	not, for example !UMLAttribute means something other than UMLAttribute
+	public
-	private
#	protected

The process we have presented appears to be a tedious process for verifying mapping correctness. We therefore present here a vision for semi-automating the task. An add-on to the prototype we propose could be built that would take a combinatorial model in a machine readable format and verify the MUT for each of the test case models/objects. The results would then be compared with the expected results. This would be an automated process that could in effect test multiple

mappings and produce a results report for the software engineer to analyze. This level of automation is a problem on its own with several challenges requiring a substantial effort to implement. For example, a Class object that contains a set of Attribute and Operation objects cannot be modeled elegantly in a combinatorial model that can be interpreted by both a human and machine. That is, it is difficult to represent sets of sets of objects in a way that can be used to automatically generate test cases and interpret test results. We reiterate here that this level of automation is future work and not currently supported by the model compliance verification prototype.

Verifying mapping completeness

This activity is concerned with verifying that all elements of the *specification* model are mapped into elements in the *tSpec* model. That is, each element of the *specification* should be used at least once during the transformation from *specification* model to *tSpec* model. In order to perform this analysis we make use of the traceability model that is produced by executing the mappings. As we described in the overview of our prototype when a set of mappings is executed on an input or source model a traceability model is generated. The traceability model is a set of traceability elements each of which includes the mapping applied, the set of elements used by the mapping (inputs) and the set of elements produced by the mapping (outputs).

```

mapping PackageMapping(package)
{
  domain {(Package)[name = n, ownedElement = els] }
  body
  {
    (Namespace)
    [
      name = n,
      classes = els->select(e | e.IsKindOf("Class"))->collect(cls | ClassMapping(cls))
    ]
  }
}

```

Figure 30 PackageMapping mapping for completeness verification example

```

mapping ClassMapping(class)
{
  domain {(Class)[name = n, feature = ftrs]}
  body
  {
    (CSharpClass)
    |
    name = n,
    attributes = ftrs->select(f | f.IsKindOf("Attribute"))->collect(a | AttributeMapping(a))
  }
}

```

Figure 31 ClassMapping mapping for completeness verification example

```

mapping AttributeMapping(attribute)
{
  domain {(Attribute)[name = n, type = t]}
  body { (CSharpAttribute)[name = n, type = t, visibility = "private"]}
}

```

Figure 32 AttributeMapping mapping for completeness verification example

Figure 33 A UML model for completeness testing example

Suppose we apply the mapping (*PackageMapping*) defined in Figure 30 to the model shown in Figure 33. We expect that each of the *Package*, *Class*, *Attribute* and *Generalization* elements contained in the model will be used in the mapping and if one is not that the set of mappings is incomplete. When the mapping is applied the traceability model shown in Figure 34 and Figure 35 is produced. Analyzing the model we are able to see that the *Generalization* elements were not used in the mappings.

```
<?xml version="1.0" encoding="utf-16"?>
<trace-info>
  <trace>
    <mapping-applied>AttributeMapping</mapping-applied>
    <source-objects><obj type="Attribute" valId="attr1" name="title" /></source-objects>
    <target-objects><obj type="CSharpAttribute" valId="catr1" name="title" /></target-objects>
  </trace>
  <trace>
    <mapping-applied>AttributeMapping</mapping-applied>
    <source-objects><obj type="Attribute" valId="attr2" name="stockCount" /></source-objects>
    <target-objects><obj type="CSharpAttribute" valId="catr2" name="stockCount" /></target-objects>
  </trace>
  <trace>
    <mapping-applied>AttributeMapping</mapping-applied>
    <source-objects><obj type="Attribute" valId="attr3" name="rating" /></source-objects>
    <target-objects><obj type="CSharpAttribute" valId="catr3" name="rating" /></target-objects>
  </trace>
  <trace>
    <mapping-applied>ClassMapping</mapping-applied>
    <source-objects><obj type="Class" valId="cls1" name="Merchandise_Item" /></source-objects>
    <target-objects>
      <obj type="CSharpClass" valId="ccls1" name="Merchandise_Item" />
    </target-objects>
  </trace>
  <trace>
    <mapping-applied>AttributeMapping</mapping-applied>
    <source-objects><obj type="Attribute" valId="attr4" name="director" /></source-objects>
    <target-objects><obj type="CSharpAttribute" valId="catr4" name="director" /></target-objects>
  </trace>
  <trace>
    <mapping-applied>AttributeMapping</mapping-applied>
    <source-objects><obj type="Attribute" valId="attr5" name="year" /></source-objects>
    <target-objects><obj type="CSharpAttribute" valId="catr5" name="year" /></target-objects>
  </trace>
</trace-info>
```

Figure 34 Traceability information produced when applying the mapping in Figure 30 to the model in Figure 33

```

<trace>
  <mapping-applied>ClassMapping</mapping-applied>
  <source-objects><obj type="Class" valId="cls2" name="Movie" /></source-objects>
  <target-objects><obj type="CSharpClass" valId="ccls2" name="Movie" /></target-objects>
</trace>
<trace>
  <mapping-applied>AttributeMapping</mapping-applied>
  <source-objects><obj type="Attribute" valId="attr6" name="system" /></source-objects>
  <target-objects><obj type="CSharpAttribute" valId="cattr6" name="system" /></target-objects>
</trace>
<trace>
  <mapping-applied>AttributeMapping</mapping-applied>
  <source-objects><obj type="Attribute" valId="attr7" name="company" /></source-objects>
  <target-objects><obj type="CSharpAttribute" valId="cattr7" name="company" /></target-objects>
</trace>
<trace>
  <mapping-applied>ClassMapping</mapping-applied>
  <source-objects><obj type="Class" valId="cls3" name="VideoGame" /></source-objects>
  <target-objects><obj type="CSharpClass" valId="ccls3" name="VideoGame" /></target-objects>
</trace>
<trace>
  <mapping-applied>PackageMapping</mapping-applied>
  <source-objects><obj type="Package" valId="pkg1" name="RoseModel" /></source-objects>
  <target-objects><obj type="Namespace" valId="ns1" name="RoseModel" /></target-objects>
</trace>
</trace-info>

```

Figure 35 Continuation of the traceability information produced when applying the mapping in Figure 29 to the model in Figure 33

Although the example we have presented here is simplistic, verifying completeness becomes a much larger task when the size of the model grows and the number of mappings needed to generate the target (*tSpec*) model increases. Through automation the prototype produces a list of each of the elements defined in the source (*specification*) model and the number of times that they are used as a source element by a mapping to produce an element in the target (*tSpec*) model. A software engineer can analyze the list for elements that are not used in the mappings and make the necessary corrections to the mappings to ensure that all objects are used. Figure 36 shows the output of the automated analysis of the traceability model. In this representation it is much easier to identify that the *Generalization* elements are not being mapped.


```
----- Completeness Info -----  
Package (pkg1) => 1  
Class (cls1) => 1  
Attribute (attr1) => 1  
Attribute (attr2) => 1  
Attribute (attr3) => 1  
Class (cls2) => 1  
Generalization (gen2) => 0  
Attribute (attr4) => 1  
Attribute (attr5) => 1  
Class (cls3) => 1  
Generalization (gen1) => 0  
Attribute (attr6) => 1  
Attribute (attr7) => 1
```

Figure 36 Results from the automated analysis of for completeness

We feel that it is important to note that the traceability model currently limits itself to those elements that are passed in as inputs to a mapping. This impacts the completeness analysis since it is possible that one of the objects used to produce a target object was selected within the body of the mapping. For example, Figure 37 presents a mapping where not all of the objects used to produce the target are captured in the traceability model. In the example, the Class element passed into the *ClassMapping* mapping would be reported in the traceability model but the Attribute elements will not be reported. Therefore, the completeness analysis will indicate that not all objects are used in the mapping execution when in fact all are. Therefore, a mapping should be applied to all objects used to produce an element in the target model. In Figure 37, a new mapping would have to be defined, that produces the target Attribute object.

```

mapping ClassMapping(class)
{
  domain {(Class)[name = n, feature = ftrs]}
  body
  {
    (CSharpClass)
    [
      name = n,
      attributes = ftrs->select(f | f.IsKindOf("Attribute"))
      ->collect(a | (Attribute)[name = a.name, type = t.type, visibility = "private"])
    ]
  }
}

```

Figure 37 Mapping where Attribute elements are used but would not be reported in the traceability model

3.1.2.2 Compliance Verification from *tSpec* to *Spec*

This activity is concerned with verifying that the mappings are complete¹⁸ and correct¹⁹ and was discussed in the previous section. Therefore, we present here a complementary approach to verify that the mappings are correctly defined. It is possible within the tool to define a set of compliance rules that will ensure that every element that appears in the *specification* does in fact appear in the *tSpec* model. This is indirectly accomplished in the prototype by creating a project specifically for this task, where the *tSpec* model plays the role of the *design* model in the project and the *specification* model plays the role of the *tSpec* model. The set of compliance rules defined within the project will check that the *tSpec* (*design*) conforms to the *specification* (*tSpec*). More specifically, every element in the *specification* model appears in the *tSpec* model.

We reiterate, however, that the completeness analysis discussed in the previous section also covers the compliance verification from *tSpec* to *specification*. The process discussed in the previous paragraph would only reinforce the confidence in the completeness and correctness of the

¹⁸ Complete implies every element in the specification model being used as the source for at least one element in the *tSpec*.

¹⁹ Here we are not referring to the formal definition instead the term ‘correct’ implies that the mapping produces the expected output for a given input.

mappings. The actual definition of such compliance rules is driven by the defined mappings. That is, the domain(s) of the mappings define which elements of the *specification* model to select; therefore, what remains is to define the domain for the *tSpec* model elements and to define the relationship between the elements from the two models. For example, the domains of the compliance rule *UMLClassWithCSharpClass* defined in Figure 40 are derived from the mapping *UMLClassToCSharpClass* (Figure 38) that it is intended to verify.

```

mapping UMLClassToCSharpClass(class)
{
  domain {(UMLClass){name = n, attributes = a}}
  body
  {
    (CSharpClass)
    [
      name = n,
      features = a->collect(att | UMLAttributeToCSharpAttribute(att))->flatten()
    ]
  }
}

```

Figure 38 Mapping definition that produces a CSharpClass from a UMLClass

The criteria for completeness we defined in the previous section (Section 3.1.2.1), requires that all elements of the *specification* model be used in at least one mapping in creating the *tSpec* model. This however, will not reveal all potential problems in the set of mappings. Suppose that some elements from the *specification* model should map into two elements of the *tSpec* model. For example, *public* and *protected* attributes in the mapping defined in Figure 39. Suppose that the verification rules for this mapping are those defined in Figure 40. The model produced will not satisfy the rules and therefore the mapping is incorrect, however, the completeness criteria would still be satisfied as all elements are used in the generation of the *tSpec*.

```

mapping UMLAttributeToCSharpAttribute(attribute)
{
  domain {(UMLAttribute) [ name = n, type = t, visibility = v ]}
  body
  {
    if v = "public" then
      Set{
        (CSharpAttribute) [ name = n, type = t, visibility = "private" ],
        (CSharpProperty) [ name = n, type = t, visibility = v, modifiable = true ]
      }
    else
      Set{
        (CSharpAttribute) [ name = n, type = t, visibility = "private" ]
      }
    endif
  }
}

```

Figure 39 Redefinition of the mapping defined in Figure 28.

```

relation UMLClassWithCSharpClass
{
  domain { (UMLClass) [ name = n, attributes = a ] }
  domain { (CSharpClass) [ name = n, features = f ] }
  when
  {
    let
      publicProtected = a->select(att | att.visibility = "public" or att.visibility = "protected")
    in
      a->forAll(att | UMLAttributeWithCSharpAttribute(att, f))
    and
      publicProtected->forAll(p | UMLAttributeWithCSharpProperty(att, f))
  }
}

```

Figure 40 Rule for verifying compliance between input and output of the UMLClassToCSharpClass mapping, supported by the relations in Figure 41 and Figure 42

```

relation UMLAttributeWithCSharpAttribute
{
  domain {(UMLAttribute) [ name = n, type = t ]}
  domain {(CSharpAttribute)[name = n, type = t, visibility = "private"]}
}

```

Figure 41 Rule for compliance between a UMLAttribute and a CSharpAttribute

```

relation UMLAttributeWithCSharpProperty
{
  domain {(UMLAttribute)[name = n, type = t, visibility = v]}
  domain {(CSharpProperty) [name = n, type = t, visibility = v]}
  when { v = "public" or v = "protected" }
}

```

Figure 42 Rule for compliance between a UMLAttribute and CSharpProperty

We note, unit testing a mapping will likely catch many of the problems that compliance rules between the *specification* model and the *tSpec* model will reveal²⁰. However, there may be faults when mappings interact with one another and creating relations to verify the mappings will help to ensure confidence in the *tSpec* model produced by the mappings. Again this activity is not currently a primary activity in the process but nonetheless the tool can support it in an indirect way.

3.1.2.3 Compliance Verification from Design to tSpec

The activity of compliance verification from *design* to *tSpec* involves verifying conformity between the *design* model and the transformed *specification* model (*tSpec*). More specifically, the activity analyzes the *design* model to ensure that it contains all of the elements defined in the *tSpec* model.

---

²⁰ As discussed in Section 3.1.2.1.

The reader is reminded that at this step of the compliance verification process the two models should be written in the same modeling language²¹.

The prototype tool, through the definition of QVT relations, supports the activity of compliance verification from *design* to *tSpec*. More specifically, a set of QVT relations defines the semantics for a *design* model to be compliant with the *tSpec* model and thus indirectly with the *specification* model. Recall, a QVT relation establishes a relationship between two or more models through the use of patterns and constraints. In the prototype we restrict relations to two input models, a *tSpec* model and a *design* model. Similar to the *specification* to *tSpec* mapping definitions, a project contains a relation that starts the compliance analysis, as well as, a set of supporting relations. The supporting relations allow the software engineer to logically structure the relations and to create relations that deal with specific types of elements in the models.

The evaluation of the compliance relations produces either a success result or failure result. If the evaluation produces a success result then the *design* model complies with *tSpec* as set forth by the compliance criteria defined in the QVT relations. For example, suppose we apply the *CSharpClassCompliance* compliance rule defined in Figure 45 and supported by *CSharpAttributeCompliance* (Figure 46) and *CSharpPropertyCompliance* (Figure 47) to the *tSpec* and *design* models shown in Figure 43²². Since this is a small model we can determine by visual inspection that the *design* does in fact comply with the *tSpec*. The prototype produces a success result since the *design* model satisfies the compliance rules.

---

²¹ This is a property of the compliance verification process and not of the prototype. The prototype can support compliance rules between a *tSpec* and design written in different modeling languages.

²² Although this example seems trivial, we remind the reader that the prototype does not know that the models are identical and that *tSpec* model is an intermediate model that has been generated by a set of transformations and that the *specification* model is not likely to be identical to the *design* model.

Figure 43 A design that is compliant with a tSpec

If the evaluation of the compliance rules fails then the *design* model does not comply with the *tSpec* model based on the semantics of compliance defined in the QVT relations. The prototype makes an effort to provide the software engineer with information on the element of the *tSpec* that does not have a corresponding compliant element in the *design*. A failure produces a message stating the relation that did not hold, as well as, the element from the *tSpec* model that the relation was trying to match in the *design* model. For example, applying the compliance criteria defined by *CSharpClassCompliance* (Figure 45) to the *tSpec* and *design* models shown in Figure 44 results in a failure. The *design* model does not have the "Age" property specified in the *tSpec*, which causes the failure. The tool produces the output shown in Figure 48.

Figure 44 A *design* that does not comply with the *tSpec*

```

relation CSharpClassCompliance
{
  domain { (CSharpClass){name = n, features = fSpec} }
  domain { (CSharpClass){name = n, features = fDesign} }
  when
  {
    let
      attributes = fSpec->select(f | f.IsKindOf("CSharpAttribute"))
      properties = fSpec->select(f | f.IsKindOf("CSharpProperty"))
    in
      attributes->forAll(a | CSharpAttributeCompliance(a, fDesign))
      and
      properties->forAll(p | CSharpPropertyCompliance(p, fDesign))
  }
}

```

Figure 45 Rule defining compliance between two CSharpClass classes


```

relation CSharpAttributeCompliance
{
  domain { (CSharpAttribute)[name = n, type = t, visibility = v]}
  domain { (CSharpAttribute)[name = n, type = t, visibility = v]}
}

```

Figure 46 Rule defining compliance between two CSharpAttribute classes

```

relation CSharpPropertyCompliance
{
  domain { (CSharpProperty) [name = n, type = t, visibility = v, modifiable = m] }
  domain { (CSharpProperty) [name = n, type = t, visibility = v, modifiable = m] }
}

```

Figure 47 Rule defining compliance between two CSharpProperty classes

```

----- Compliance FAILURE -----
<?xml version="1.0" encoding="utf-16"?>
<trace-info>
  <trace-item mapping="CSharpClassCompliance" e-name="Person"
    e-type="CSharpClass" e-id="95ccad02b615" result="FAIL">
    <children>
      <trace-item mapping="CSharpPropertyCompliance" e-name="Age"
        e-type="CSharpProperty" e-id="cee3633f3b81" result="FAIL">
        <children />
      </trace-item>
    </children>
  </trace-item>
</trace-info>

```

Figure 48 Example of results produced when compliance fails

The information produced by the compliance failure assists the software engineer in determining the element from the *tSpec* model that could not be found in the *design* model. In Figure 48 the information states that the rule *CSharpClassCompliance* failed when trying to find a *CSharpClass* element named *Person*. However, this traceability item has a child element that caused it to fail; a *CSharpProperty* named *Age*. The software engineer can then locate the root element (*Person:CSharpClass*) and in the *design* and see that it is missing or incorrectly defines the *Age*

property. If one of the properties of an element fails then that element is also labelled as having failed; for example the *Person* element in our previous example. Adding the *Age* property to the *Person* element and re-evaluating the compliance rules should result in successful compliance. We note that, when more than one element in a set (from the *tSpec* model) is not found in the *design* then each one of those elements is listed as having failed.

3.1.3 Revisiting our objectives

In this section we revisit each of the objectives that we defined in the beginning of the chapter. For each objective the prototype components that realize the objective are stated and discussed.

Objective 1 – The ability to import, define, manipulate and save specification, tSpec and design models

The prototype provides the ability to load and save models through the Project Explorer and Model Inspector components. These models can either be loaded from a model that the tool has previously saved or a Model Converter can be used to extract the model from a third-party tool such as IBM Rational Rose. Once a model has been loaded, it can be manipulated through the appropriate Model Inspector. This component provides the ability to add, modify and delete the elements of a model. A model is saved to an XML file following the XML Schema [67] shown in Appendix D. This saved model is then reloaded when the project is reopened.

Objective 2 – The ability to define and easily modify mappings to transform the specification into a semantically similar specification in the design modeling language;

The QVT language is the OMG effort to define an industry standard for model mappings. In the prototype we use the mapping language defined in the most current (at the time of writing) QVT submission by the QVT Merge group [31]. The prototype provides a text editor for editing mappings, which can then be associated with a Model Compliance Verification project. Each mapping used in the generation of the *tSpec* model is defined in its own flat file. The changes made to a mapping definition are taken into account by re-executing the mappings. This does not require recompiling any component of the prototype or any other tool.

The mapping language does not bind the software engineer to any particular metamodel or modeling language. The only requirement is that the language or metamodel used for a particular model be well defined. This is a consequence of choosing the QVT mapping language.

Objective 3 – The ability to define and easily modify compliance rules to verify that a design model is compliant with a tSpec model

Again this objective is addressed by the prototype through the use of the QVT relation definition language, which is closely related to the mapping definition language. The same text editor is used for relations and for mappings. The relations can be linked to a project in its compliance rules section. Like the transformation definition language the compliance rule (relation) definition language does not bind the software engineer to any particular modeling language or metamodel for their *specification* and *design* models.

Like mappings modifications to compliance rules do not require recompiling the prototype or any of its components. The modifications can be made through the text editor provided in the IDE and are taken into account the next time the compliance rules are evaluated.

Objective 4 – The ability to execute the defined mappings on a specification model to produce the tSpec model

As stated in the discussion of Objective 2 mappings are defined using the QVT mapping definition language. Therefore, the QVT engine enables the execution of the mappings defined to create the *tSpec* model. The QVT engine is an interpreter that is able to understand QVT mapping definitions and to produce a target model or model element based on the semantics defined in [31]. The model or element that is produced by executing the mappings is shown in the *tSpec* Model Inspector and can be saved through the IDE.

Objective 5 – The ability to answer the question “Is my design model compliant with the specification model?” by applying the compliance rules to the tSpec and design models

Again as stated in the discussion of Objective 3 compliance rules are defined using the QVT relation definition language. Therefore, the QVT engine component enables the prototype to

answer the compliance question between a *design* and a *specification*. By evaluating²³ the QVT relation definitions that define the semantics of compliance the prototype is able indicate to the software engineer whether the *design* is compliant. When a *design* is not compliant the prototype tries to guide the identifying the elements of the *specification* that are not properly realized in the *design*. This is achieved by the trace information that is produced when compliance fails, showing the elements in the *tSpec* for which no match was found in the *design*.

3.2 Comparison to an existing model compliance verification tool

This section compares the prototype to a tool that is discussed in [70]. The section first presents the capabilities of the tool from [70] which we call the MCV tool. This is followed by a discussion of the factors that differentiate our prototype from that tool.

3.2.1 The MCV tool

The tool or set of tools discussed and developed in [70] support the activity of model compliance verification in a specific context, name the Software Radio Components specification, IBM Rational Rose and RoseRT. The MCV tool parses an Rose model file, which is the *specification* model to create an internal representation of the model. While parsing the model file the tool logs those lines of the model file that were not used as information in the internal model. As the model is parsed a tree structure is created for the internal model representation.

A set of hard coded mappings is applied to the internal representation of the *specification* model to generate a *tSpec* model. These mappings are considered hard coded since they are written in the C++ programming language [59] and compiled with the tool. That is, the *specification* to *tSpec* mappings cannot be modified without recompiling the tool. The resulting *tSpec* model is output to a Rational Rose Real-Time model file.

---

²³ The operational semantics for evaluating a QVT relation are defined in QVT Merge submission.

In order to verify mapping correctness the tool has built into it criteria for ensuring that all test cases are covered for a mapping. These criteria are derived from OCL statements associated with the mappings. The OCL statements were used to manually build a combinatorial model to verify the correctness of the mapping. When the tool is executed on a test model it verifies that at least one element in the model has satisfied each row of the combinatorial model. This verifies that a test source model provided a complete test suite for a mapping.

Upon completion of the *specification* to *tSpec* mappings the MCV tool checks for semantic equivalence between the two models. This is achieved by verifying that every element in the *specification* model is used at least once as the source element of a target element found in the *tSpec*. When a *specification* model element is not used in generating the *tSpec* the tool indicates its type and location in the *specification* model. As a consequence the tool is able to generate traceability information, which indicates which *specification* model elements were used in creating a particular *tSpec* model element.

The MCV tool completes its execution by parsing a IBM Rational Rose Real-Time model (*design* model) into its internal format. This *design* model is then compared to the internal representation of the *tSpec* model and any discrepancies are identified. We note, that the MCV tool restricts the elements it considers in both the *specification* and *design* models to those elements that are relevant to the SCR [44]. The tool also addresses many of the challenges we identify in Section 3.3, however, again they are addressed in a hard coded manner.

3.2.2 Prototype vs. MCV: a comparison

This section differentiates our proposed prototype from the MCV tool discussed in the previous section. The major difference between the two is that the prototype discussed in this dissertation is open. More specifically, the software engineer is not restricted to any particular modeling languages or metamodels. Furthermore, the prototype provides the ability to modify and re-execute model-to-model transformations in order to create the *tSpec* model. Whereas the MCV tool is specific to the UML 1.5 and RoseRT metamodels: it cannot modify the mappings without editing the C++ source and recompiling.

Our prototype also allows for a more flexible compliance verification process. For example, the software engineer is not required to create a *tSpec*; they can create compliance rules between models in different modeling languages with well-defined metamodels. More specifically, it is possible to bypass the definition of *specification* model to *tSpec* model mappings and define compliance rules between the *specification* and *design* model. The necessity of the *tSpec* model was driven by the semantic gap between modeling languages and the goal of the *tSpec* is to bridge this gap. Within our prototype when the *tSpec* model is bypassed the language bridge is embedded into the definition of the compliance rules.

The MCV tool does a good job at ensuring that the set of mappings it has defined both produces the correct target (correctness) and uses every element in the source model in producing the target model (completeness). This is a consequence of knowing the exact elements and mappings that the tool can understand and execute. When the modeling language and mappings or compliance rules are not known at compile time it is a more difficult task to ensure correctness and completeness. We have, however, shown in Section 3.1.2.1 that it is possible to achieve similar results in our more generalized prototype. With better support within the prototype verifying the correctness mappings and compliance rules could become less tedious. That is, by providing better semi-automated and automated support for verifying mappings and compliance rules we reduce the effort required by the software engineer.

Another artefact produced by the MCV tool that is not produced by our prototype is a classification of elements in the *specification* and *design* models. The MCV tool produces a report after evaluating compliance, which classifies each element in the two models as either *specification* only, *design* only or common.

specification only element – an element that appears in the *specification* but no compliant element is found in the design;

design only element – an element that appears in the design but not in the *specification*;
and

common elements – elements in the *specification* that have compliant counterpart in the design and elements in the design that are compliant counterparts of *specification* elements.

Our prototype produces a report of specification only elements, if they exist. A *specification* only element exists if the *design* model is not compliant with the *specification* model. That is, if the evaluation of the compliance rules fails a *specification* only element exists, since it cannot be found in the *design* model. It is possible to determine *design* only elements in the prototype by switching the *design* and *tSpec* models in the project and re-evaluating the same compliance rules. It is then possible to infer that those elements that do not appear on either list are common elements.

One last major difference between the prototype and the MCV tool is that, in our work, the semantics of compliance are defined in a standard language. That is, the prototype forces the software engineer to define the compliance rules in the QVT relation definition language. This has several benefits; the most important being that the semantics of compliance will be unambiguous. More specifically, the QVT language has a well-defined and unambiguous syntax and semantics [31], which make the QVT compliance rules unambiguous. Furthermore, as with the mappings a software engineer can modify the compliance rules and re-evaluate them within the prototype.

We feel that this comparison shows that there are advantages to the use of a standardized mapping and compliance rule definition language like QVT.

3.3 Addressing the Challenges

In this section we address some of the challenges in model compliance verification that we defined in [71]. The focus is on a non-exhaustive set of challenges that we feel are general to most model compliance verification efforts. It is recognized that each specific effort will face unique challenges due to many different factors including domain and modeling languages. The challenges that are addressed include: model representation language, type correspondence, name

correspondence, optional *specification* elements, realizing a subset of a *specification*, realizing multiple specifications and package structure.

3.3.1 Representation Language

There is a possibility that the *specification* and *design* models use different modeling languages or metamodels. The solution to this particular challenge is addressed through the two-step process described in [70] and reviewed in Section 2.3. That is the *specification* model is transformed into a semantically similar model (*tSpec*) expressed in the *design* language. In the prototype tool this particular challenge is addressed by the definition of the QVT mappings that create a *tSpec* model from the *specification* model.

We recognize that the use of QVT makes this challenge less of an issue. More specifically, it is possible to define QVT relations to verify compliance between two models that are expressed in different modeling languages. The QVT relation concept removes the absolute necessity of the *tSpec* model. However, in developing the prototype we felt that it was best to retain the two separate activities for several reasons including:

- Providing a clear separation between the verification of compliance and relating the *specification* language elements with the design language elements; and
- Simplifying the task of identifying the elements that are missing from the design through the approach discussed in Section 3.1.2.3.

Let us elaborate. In our research we have found that when writing compliance rules for models expressed in different modeling languages each with many concepts (e.g. UML and RoseRT) the compliance rules become complex quickly. Allowing the software engineer to separate the task of creating the mappings to transform the *specification* language elements into *design* language elements and the task of defining the compliance rules between a *specification* model and a *design* model produces artefacts (mapping definitions and compliance rule definitions) that are easier to understand and maintain. That being said, in the prototype there is nothing stopping the software

engineer from bypassing the creation of the *tSpec* and directly defining compliance rules between a *specification* model and *design* model that use different modeling languages.

The product of compliance verification within the tool is a *success* or *failure* result (as discussed in Section 3.1.2.3). When a *design* is non-compliant we produce a trace of the relations that could not be satisfied as well as the particular element in the *tSpec* (or, if bypassed, the *specification*). When the *specification* element that is missing from the *design* is expressed in the *design* language it should provide the software engineer with more guidance in making the necessary modifications to the *design* than the same element expressed in the *specification* language.

3.3.2 Type Correspondence

The *specification* and *design* models may use different names for abstract types when referring to the same primitive type. For example, while some languages may use the type *Integer* to define a 32-bit integer value, some others may refer to the same primitive type as *int*. Figure 49 presents an example of the type correspondence challenge, the *design* will not comply with the *specification* because it uses *int* rather than *Integer* even though they have the same semantics.

Specification	Design
Manager name : String age : Integer	Manager name : string age : int

Figure 49 Type correspondence example

Our proposed solution makes use of the additional information model in the prototype. Recall from Section 3.1.1.1, that the additional information model allows the software engineer to

provide a second input model with information that can be used when defining the mappings from *specification* to *tSpec*. The solution adds a set called *TypeCorrespondence* to the additional information model; each element of the set is a *TypeConversion* object. A *TypeConversion* object contains a *source* type and a *target* type. The *source* identifies the data type name used in the *specification* model while the *target* type identifies the name of the corresponding type in the *design*. For example, suppose the *specification* language uses **String** and **Integer** while the *design* language uses **string** and **int** respectively, the additional information model would resemble the one presented in Figure 50.

The *typeConversion* objects in the additional information model are used to convert type names in the *specification* model into their corresponding *design* language type. This is done during the creation of the *tSpec*, more specifically within the mappings that produce the *tSpec*. For example, in the mappings defined in Figure 51 and Figure 52, the additional information is passed into the *TypeCorrespondence* mapping, which selects the type correspondence set, to pass into the *AttributeToAttribute* mapping. The set is then checked to see if the type needs to be converted in the *tSpec*. If so it converts it, otherwise the same type name is carried through.

Figure 50 Example of the additional information model for solution to type correspondence challenge

```

mapping TypeCorrespondence(cls, additional)
{
  domain { (Class)[ name = n, attributes = att ]}
  domain { (AdditionalInfo)[ TypeCorrespondence = tc ]}
  body
  {
    (Class)[ name = n,    attributes = att->collect(a | AttributeToAttribute(a, tc)) ]
  }
}

```

Figure 51 Mapping a Class to a Class incorporating the type correspondence information

```

mapping AttributeToAttribute(att, typeInfo)
{
  domain { (Attribute)[name = n, type = tj }
  body
  {
    let
      ctype =
        if typeInfo->select( ti | ti.IsKindOf("TypeConversion") and ti.source = t )->notEmpty() then
          typeInfo->select( ti | ti.IsKindOf("TypeConversion") and ti.source = t )->first().target
        else
          t
        endif
    in
      (Attribute)[ name = n, type = ctype ]
  }
}

```

Figure 52 Mapping an Attribute to an Attribute making use of the type correspondence information

This proposed solution addresses the challenge during the transformation from *specification* to *tSpec*. The *design* types are then checked against the *tSpec* during the evaluation of *design* compliance with the *tSpec*. In our example (Figure 49), given the compliance rules defined in Figure 53, the *design* would be compliant with the *tSpec* produced by the TypeCorrespondence mapping and therefore compliant with the *specification*. An alternative to our proposed solution would be to embed the type conversion into the mappings. That is, the mappings where type conversion is required would contain a set of *if...then...else* expressions to convert the type. The advantage of our proposed solution is that have localized the type conversion information to the AdditionalInformation model, therefore additions, subtractions or modifications to the type conversions is localized to the AdditionalInformation model.

```

relation ClassCompliance
{
  domain { (Class)[ name = n, attributes = attSpec ]}
  domain { (Class)[ name = n, attributes = attDesign ]}
  when
  {
    attSpec->forAll(a | AttributeCompliance(a, attDesign))
  }
}

relation AttributeCompliance
{
  domain { (Attribute)[ name = n, type = t ]}
  domain { (Attribute)[ name = n, type = t ]}
}

```

Figure 53 Type correspondence example compliance rules

3.3.3 Naming and Role Playing

An element in the *design* model may realize an element in the *specification* model but use a different name. For example, Figure 54 presents the Manager class in the *specification* model, which is realized by the BankManager class in the *design* model. A search for the Manager class in the *design* model will prove unsuccessful because no class exists with the same name.

Figure 54 Naming and role-playing example

As one possible resolution to this particular challenge of model compliance we propose providing additional information to the mappings that produce the *tSpec*. Similar to the type correspondence challenge (Section 3.3.2) we add a set *NameCorrespondence* to the additional information model. The set contains elements of type *NameConversion* that define *specName*, *designName* and *scope* properties. The *specName* property is the name in the *specification* model while the *designName* property is the corresponding name in the *design* model. The *scope* property defines the types of elements to which the renaming applies. For example, the additional information model in Figure 55 provides the necessary information for our example in Figure 54.

Figure 55 Example of the additional information model for solution to name and role playing challenge

With the additional information model we convert names used in the *specification* model into their corresponding name in the *design* model. For example, in the mappings defined in Figure 56 and Figure 57 the additional information is passed into the *NamingAndRolePlaying* mapping, which selects the name correspondence set. The set is then checked to see if the name needs to be converted in the *tSpec*, if so it converts the name, otherwise the same name is carried through. We note that we have focused specifically on the naming of objects, however, a similar pattern of defining additional information and using it the mappings can be used to rename any element in the *specification* model.

```

mapping NamingAndRolePlaying(cls, additional)
{
  domain {(Class)[ name = n, attributes = att ]}
  domain { (AdditionalInfo)[NameCorrespondence = nc]}
  body
  {
    let
      cname =
        if nc->select( cn | cn.specName = n and cn.scope = "Class" )
          ->notEmpty() then
            nc->select( cn | cn.specName = n and cn.scope = "Class" )
              ->first().designName
          else
            n
          endif
    in
      (Class) [ name = cname, attributes = att->collect(a | AttributeToAttribute(a)) ]
  }
}

```

Figure 56 Mapping a Class to a Class incorporating the name correspondence information

```

mapping AttributeToAttribute(att, typeInfo)
{
  domain {(Attribute)[ name = n, type = t ]}
  body
  {
    (Attribute)[ name = n, type = t ]
  }
}

```

Figure 57 Mapping an Attribute to an Attribute making for the name correspondence example

This proposed solution addresses the challenge during the transformation from *specification* to *tSpec*. The *design* names are then checked against the *tSpec* during the verification of *design* compliance with the *tSpec*. In our example (Figure 54), given the compliance rules defined in Figure 53, the *design* would be compliant with the *tSpec* produced by *NamingAndRolePlaying* and therefore compliant with the *specification*.

3.3.4 Mandatory vs. Optional Elements

The *specification* model may state that only a subset of the elements it defines are required (mandatory elements) to be in the *design* model, while the remaining elements are considered optional elements. More specifically, a designer may or may not decide to realize the optional elements. For example, in Figure 58 only the *PremiumSavings* class is optional therefore the *design* model is compliant. Those *specification* elements that are optional must be clearly indicated so that the model compliance verification prototype can take this into consideration.

Figure 58 Mandatory vs. optional example

The proposed resolution of this challenge is to provide additional information to the *tSpec* creation mappings. This kind of additional information states those elements that are optional, as well as, whether they are realized in the *design*. This is achieved by adding a set, *Optional*, to the additional information model. The set contains *OptionalElement* objects that are defined by three properties *type*, *name* and *realized*.

type – the type of element that is optional, for example Class or Attribute;

name – the name of the optional element, for example PremiumSavings; and

realized – a Boolean value that when true indicates that the element is intended to appear in the design and when false the element has not been realized in the design

The additional information model presented in Figure 59 is an example of the additional information required for our optional element example presented in Figure 58. The example has a single optional element named *PremiumSavings* of type *Class* that was not realized.

Figure 59 Example of additional information used in addressing the optional element challenge

The optional element information that is added to the additional information model makes explicit those elements that are optional and furthermore those optional elements that have been realized in the *design*. The optional element information can now be incorporated into the mappings that create the *tSpec*. That is, we only map those elements that are either mandatory, or optional and realized in the *design*. This particular resolution assumes that if an element is not a member of the optional element set then it is a mandatory element. For example, Figure 60 presents a mapping that is applied to *Package* elements that will only map those classes that are either mandatory (i.e. not labelled as optional) or are optional and realized.


```

mapping OptionalElements(package, additional)
{
  domain {(Package)[ name = n, ownedElement = owned ]}
  domain {(AdditionalInfo)[ Optional = optionalEls ]}
  body
  {
    let
      classes = owned->select(el | el.IsKindOf("Class") and
        optionalEls->select(opt | opt.type = "Class" and opt.name = el.name and opt.realized = false)
        ->isEmpty())
    in
      (Package)[ name = n, ownedElement = classes->collect(cls | ClassToClass(cls, optionalEls)) ]
  }
}

```

Figure 60 Mapping that only maps optional *Class* elements that have been marked as realized

Applying the *OptionalElements* mapping defined in Figure 60 to the *specification* in our example, produces the *tSpec* shown in Figure 61. Therefore, the *design* is compliant with the *specification*, since *PremiumSavings* is optional and the software engineer decided not to realize it (Figure 59). Should the software engineer decide that *PremiumSavings* is to be realized by the *design* they need only change the *realized* value in the additional information model to true and regenerate the *tSpec* model. That is, no modifications to the mapping definitions are required to take into account the realization of optional elements.

Figure 61 A compliant design taking into account the optional specification elements not realized

The proposed solution to the optional vs. mandatory element challenge we have presented in this section addresses optional elements during the *tSpec* creation activity. While the example presented

addresses only elements of type Class, the solution can incorporate other types of elements simply by passing the optional element information to the appropriate mapping and making use of it within that mapping. Furthermore, when it is decided that optional elements should now be realized, changes are localized to the additional information model and do not require modifications to either the mappings or the compliance rules.

3.3.5 Designs Implementing a Subset of a Specification

This challenge is similar to the Mandatory vs. Optional challenge of section 3.3.4, the main difference is that the designer has the ability to verify compliance with only a subset of the *specification* model elements, which we call the target elements. Therefore, when assessing a *design* model for compliance we are only interested in those elements of the specification model that are target elements. For example, consider the *specification* and *design* in Figure 62, and assume that the designer has selected only those elements in the *Investments* package as target elements. In this case, the *design* can be deemed as compliant. If no target elements are specified then it is assumed that all mandatory elements in the *specification* must appear in the *design* and in our example the *design* would be deemed as non-compliant.

Figure 62 Implementing a subset of the specification

This challenge is resolved in a similar manner to the resolution of the optional vs. mandatory challenge. That is, we add to the additional information model those elements that are not to be included in the compliance verification. For example, the additional information model in Figure 63, states that the *Package* element with name *Accounts* should be excluded from compliance verification. The mappings that create the *tSpec* would not map the elements contained within the *Excluded* set in the additional information model.

Figure 63 Example of an additional information model for resolving the subset challenge

This proposed resolution to the *specification* model subset challenge addresses the challenge during the transformation from *specification* model to *tSpec* model. That is, those model elements included in the *Excluded* set will not be mapped into the *tSpec*. The advantage to this approach is that it does not require redefinition of the mappings or the compliance rules when verifying compliance of the complete *specification* or only a subset of the *specification*.

3.3.6 Multiple-roles

Having a software element in the *design* play the role of two or more elements in the *specification* corresponds to a *design* decision and should be taken into account when assessing a *design* for compliance. The implications of this challenge are similar to those presented in the Naming and Role-Playing challenge (Section 3.3.3). An additional problem arises if two or more of the *specification* elements that a single *design* element realizes contain similar properties (for example an attribute with the same name). It is more than likely that the designer will be forced to rename one of the *specification* properties in the *design* unless they are identical. This can be captured by naming correspondence information in the additional information model and taken into account by the *specification* to *tSpec* mappings.

As this challenge is similar to the Naming and Role-playing challenge we re-use its proposed solution to resolve this challenge. For example, suppose we have two *specification* classes *ClassA* and *ClassB* that are both realized by *ClassDesign* in the *design* model. In the naming correspondence information we provide the correspondence between *ClassA* and *ClassDesign* and *ClassB* and *ClassDesign*, as shown in Figure 64. During the *specification* to *tSpec* mappings both *ClassA* and *ClassB* will be renamed to *ClassDesign* and the compliance rules will verify that *ClassDesign* complies with both.

This resolves part of the challenge; there is the possibility that for example both *ClassA* and *ClassB* have an attribute with the same name. Again, in the *design* model the software engineer is forced to rename one of them. Therefore, the additional information model will include information to rename at least one of the Attributes. If the software engineer forgets to add the necessary renaming information to the additional information model, compliance will fail since at the most one of the Attribute elements will have a match in the *design* model. We note that it is possible that both *ClassA* and *ClassB* define an Attribute element with the same properties, which can be realized in the *design* by one Attribute element. In this case it is not necessary to rename one of the two Attributes.

Figure 64 Example of additional information for the multiple-role challenge

The proposed resolution to this challenge is addressed during the creation of the *tSpec* from the *specification*. That is, the mappings from *specification* to *tSpec* incorporate the additional information for multiple-role players in the *design*. This does not require any additional statements or instructions in the mappings other than those provided for the Naming and Role-playing challenge. Furthermore, this resolution does not require modifications to either the mappings or the compliance rules when the multiple-role player information changes. Such changes are localized to the additional information model.

3.3.7 Design implementing more than one specification

In some situations a *design* model may realize more than one *specification*. In other words, in this situation, the *design* must be compliant with two *specifications*. We do not feel that this is a major issue, as the challenges we have addressed to this point provide the necessary mechanisms for handling compliance with multiple *specifications*. For example, in realizing multiple *specifications* it is highly unlikely that a designer will be able to adhere to the package structure of both *specifications*. In such a situation the designer would make use of the solution we have described in section 3.3.8.

Our general approach to this challenge would be to check the *design* for compliance with one *specification*. This may require extensive use of the solutions to the challenges we have presented. Once the first *specification* has been complied with, the designer can verify compliance with the second *specification*. In Figure 65, we illustrate the decision process for deciding if a *design* complies with two *specifications*. Initially, the same *design* is verified for compliance with each *specification*. If the *design* complies with both *specifications* then it is deemed compliant, otherwise it is non-compliant.

Figure 65 Decision process for verifying compliance with multiple specifications

We note, there were no specific modifications to the tool in order to support compliance with multiple *specifications*. The designer is required to create two projects, one for each *specification* and to perform the verification independently. Supporting multiple *specifications* in a single model compliance verification project is feasible. However, we leave this to future work as it is still possible obtain the same result with the tool as it is.

3.3.8 Package Structure

Packages are useful for organizing model elements into logical groupings. It may be an easy task to verify that a *design* follows the same package structure as the *specification*. However, we believe that it is not always necessary for a *specification* and *design* to have identical package structures to be compliant. A designer must specify whether they intend to adhere to the package structure of the *specification* or not. We note that in a *specification* the package structure may carry semantics and therefore must be followed, in which case the designer has no choice. For example, in Figure 66 the designer has intentionally chosen to create his or her own package structure.

Figure 66 Package structure example

The compliance rules must be written so that package structure is removed from both the *tSpec* model and the *design* model. Suppose we wish to define compliance rules for the *tSpec* model and *design* model shown in Figure 67, where it has been decided package structure is not significant. Our proposed approach is to first remove that Package elements from the model in the compliance rules. This results in a flattened model where all of the Class elements have been collected together. We can then reuse the same Class element compliance rule that we would have if we had chosen to comply with the package structure.

Figure 67 Example tSpec model and design model for Package structure challenge

In Figure 68 we define a ModelCompliance rule that ignores the package structure. It first flattens all of the *tSpec* classes to a single set using the CollectClasses mapping and then all of the *design* classes to a single set using the same CollectClasses mapping. The rule then iterates through each of the *tSpec* classes to verify that a compliant *design* class exists in the *design* class set. We have not shown the ClassCompliance rule since it is not affected by our proposed solution to this challenge. We note that this challenges demonstrates the flexibility in using a transformation and conformance rule definition language like QVT.

```

relation ModelCompliance
{
  domain { (Package) [ classes = tCls, packages = tPkgs ] }
  domain { (Package) [ classes = dCls, packages = dPkgs ] }
  when
  {
    let
      tSpecClasses = tCls ->including(tPkgs ->collect(p | CollectClasses(p))) ->flatten()
      designClasses = dCls ->including(dPkgs ->collect(p | CollectClasses(p))) ->flatten()
    in
      tCls ->forAll(c | ClassCompliance(c, designClasses))
  }
}

mapping CollectClasses(pkg)
{
  domain { (Package) [ classes = cls, packages = pkgs ] }
  body
  {
    cls ->including(pkgs ->collect(p | CollectClasses(p))) ->flatten()
  }
}

```

Figure 68 Compliance rule that ignores the package structure in the model

Our proposed solution addresses the challenge during compliance rule definition. That is, our compliance rules define that the package structure will be ignored. Alternatively, we could have flattened the package structure during the *specification* to *tSpec* transformation step. In which case the mapping definitions would have removed the package structure. This alternative approach would still require the flattening of the design model in the compliance rules. Therefore, we felt that it was best to flatten both models in the compliance rules. We note that since the QVT engine is not optimized; the approach of flattening models, which creates somewhat large sets, is not as efficient as desired.

Chapter 4

CASE STUDY

This chapter provides a description of a case study used to demonstrate the feasibility of our prototype. This is achieved by:

1. Defining a set of QVT mappings to transform a *specification* model into a *tSpec* model;
2. Defining a set of QVT relations (compliance rules) to evaluate compliance between a *design* model and the *tSpec* model;
3. Executing the mappings to produce a *tSpec* model; and
4. Evaluating the compliance rules against a *design* model to show compliance or non-compliance.

This particular case study addresses compliance between an IBM Rational Rose Real-Time (RoseRT) [56] *design* and a UML 1.5 [45] *specification* modeled in IBM Rational Rose [56]. The UML 1.5 *specification* model is transformed into a *tSpec* model in the RoseRT language. We first provide an overview of the case study. This is followed by a description of the two metamodels, UML 1.5 and RoseRT, which is specific to the elements that are relevant to this case study. We then define and explain the necessary mappings and compliance rules for model compliance verification in the case study. The chapter concludes by discussing the evaluation of a *design* model for compliance with the case study *specification* model.

4.1 Case Study Overview

In this case study we use our prototype tool for verifying compliance between a *specification* model that uses a subset of the UML 1.5 and a RoseRT *design* model. It is our intention to show that it is feasible to use the prototype in verifying compliance between a *specification* model and *design* model written in different modeling languages. While the metamodels of the two modeling languages have many similarities, there are also some significant differences. The most important difference is the following RoseRT capsule, port and protocol concepts for which there are no semantic equivalent in UML 1.5.

For the purposes of this case study a *specification* consists of a class model including packages, classes, interfaces, attributes, operations, generalization relationships and associations. These specification model elements are influenced by an analysis of the UML 1.5 elements found in industrial specifications [70]. In order to identify those classes that are active, additional information accompanies the class model in the form of a list of class names that tag a class as active. A *specification* also includes a set of sequence diagrams that capture the interaction between the active classes in the class model. A *design* consists of a set of packages, classes, capsules and their structure, protocols, generalization relationships and associations.

The remainder of this case study first presents the relevant subset of the *specification* and *design* metamodels. A set of mapping definitions is then defined that map the UML 1.5 *specification* into a RoseRT *tSpec* model. We then present the test model for two of the mapping definitions. This is followed by the compliance rule definitions. The mappings and compliance rules are then used in evaluating a *design* model against our *specification* model. We have chosen a fictitious case study for two primary reasons: simplicity and understandability. Simplicity is important for the purposes of testing and evaluating the feasibility of our prototype. There is a downside to using the simpler specification model in place of a specification like the SRC; namely it does not show the scalability of our prototype. Secondly, an industrial standard like the SRC requires a certain level of domain knowledge, using the fictitious specification model we are able to focus on the prototype rather than the domain knowledge required to understand the case study.

4.2 Metamodels

In this section we present the subset of the metamodels for both the *design* and *specification* languages. The subset of the *specification* metamodel reflects an analysis of the elements that appear in industrial specifications like the SRC found in [70]. And, the subset of the *design* metamodel is determined by the elements that are generated by transforming the *specification* metamodel elements.

The metamodel elements are presented in a textual format where each element is presented with its features (attributes and association ends). For example, the metamodel for a language (MyUML) consists of three concepts NamedElement, Package and Class as shown in Figure 69.

Figure 69 Metamodel for MyUML modeling language

For each of the leaf elements (Package and Class) we present a table similar to the one Table 8. The Feature column is the name of the feature (attribute or association end), for example classes for the Package element. If a feature is inherited from another concept the 'Inherited From' column indicates the parent element, for example the *name* feature for the Package element. The type

column is self-explanatory and the multiplicity defines the multiplicity of the feature if it is an association end. Below a feature an explanation for it may be given, for example the parent feature in Table 8.

Table 8 Example of metamodel element explanation for the Package element in Figure 69

Element : Package			
Feature	Inherited From	Type	Multiplicity
name	NamedElement	String	
classes		Class	n
packages		Package	n
parent		Parent	0..1
If the package is contained within another package, parent specifies its container.			

4.2.1 UML 1.5

An overview of the UML 1.5 was provided in Section 2.4.3. Diagrams of the metamodel can be found in Appendix C. We repeat, the subset of elements discussed reflects an analysis of the elements that appear in industrial specifications like the SRC found in [70]. Some of the elements that are used in our case study have additional features that we do not consider. For example, the Attribute element has a visibility property that is not considered in this case study. There are also numerous elements like Constraint, ParameterizedClass, Stereotypes, etc. that are not included in this subset. These elements could potentially change the mappings we define in Section 4.3. The impact of including additional elements is an avenue that has not been explored for this dissertation and is left to future work. We do expect that adding additional elements and properties of elements to the mappings will affect the *design* metamodel elements that are produced.

We note that most elements have a '*name*' feature, which is the name given to an instance of the element. In the UML 1.5 this name feature is a Name element, however, in the IBM Rational Rose toolset it is a string therefore we define it as a string. This simplification does not change the results we obtain; its only consequence is simplifying the mappings defined in Section 2.4.3.

4.2.1.1 *Attribute*

Element : Attribute			
Feature	Inherited From	Type	Multiplicity
name	ModelElement	String	
type	StructuralFeature	String (Classifier)	0..1
<i>The type of the attribute. In the UML specification it references a classifier, however, in the Rational Rose toolset it is a string, therefore in this dissertation we consider it to be a string.</i>			

4.2.1.2 *Operation*

Concept : Operation			
Feature	Inherited From	Type	Multiplicity
name	ModelElement	String	
parameter	BehavioralFeature	Parameter	n
<i>A possibly empty list of parameters that define the operation's signature. The list includes the return parameter if it exists.</i>			

4.2.1.3 *Class*

Element : Class			
Feature	Inherited From	Type	Multiplicity
name	ModelElement	String	
feature	Classifier	Feature	n
<i>A possibly empty set of features (Attributes and Operations) owned by the Class.</i>			
generalization	GeneralizableElement	Generalization	n
<i>A possibly empty set of Generalization relationships owned by the Class. Defines the set of classes that this class specializes and also the set of Interfaces that the class realizes.</i>			

4.2.1.4 *Interface*

Element : Interface			
Feature	Inherited From	Type	Multiplicity
name	ModelElement	String	
feature	Classifier	Feature	n
<i>A possibly empty set of features (Operations) owned by the Interface. Note that the feature set of an Interface should only contain Operation elements, even though a Classifier's feature set can contain both Attribute and Operation elements.</i>			

4.2.1.5 Package

Element: Package			
Feature	Inherited From	Type	Multiplicity
name	ModelElement	String	
ownedElement	Namespace	ModelElement	n
<i>A possibly empty set of ModelElements (Classes, Interfaces, Associations and Packages) that are owned by the Package.</i>			

4.2.1.6 Generalization

Element: Generalization			
Feature	Inherited From	Type	Multiplicity
parent		GeneralizableElement	1
<i>The Class or Interface that the owning Class or Interface specializes.</i>			

4.2.1.7 Association

Element: Association			
Feature	Inherited From	Type	Multiplicity
name	ModelElement	String	
connection		AssociationEnd	2..n
<i>A non-empty set of ends that specify the participants and the properties of the Association.</i>			

4.2.1.8 AssociationEnd

Element: AssociationEnd			
Feature	Inherited From	Type	Multiplicity
name	ModelElement	String	
aggregation		String	
<i>Specifies whether this end of the Association is an aggregation with respect to the other end of the Association. In the UML metamodel this is an Enumeration however, the Rational Rose toolset specifies it as a string therefore we use it as a string. Possible values are "none", "aggregate" and "composite".</i>			
participant		Classifier	n
<i>A mandatory property that specifies the Classifier participating in the Association at this end of the Association.</i>			

4.2.1.9 Parameter

Element: Parameter			
Feature	Inherited From	Type	Multiplicity
name	ModelElement	String	
kind		String	
<i>The kind of Parameter. In the UML metamodel this is an Enumeration however, the Rational Rose toolset specifies it as a string therefore we use it as a string. Possible values are "in", "out", "inout" and "return". We are only concerned with this property for retrieving the return type of an operation.</i>			
type		String	1
<i>The type of the Parameter. In the UML this is of type Classifier, however, in the Rational Rose toolset it is a string, therefore we use it as a string.</i>			

4.2.1.10 Interaction

Element: Interaction			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
messages		Message	n
<i>The messages exchanged in the Interaction</i>			

4.2.1.11 Message

Element: Interaction			
Feature	Inherited From	Type	Multiplicity
receiver		Class	1
<i>The receiver of the message</i>			
sender		Class	1
<i>The sender of the message</i>			
operation		Operation	0..1
<i>The operation that the message references</i>			
isOperation		Boolean	
<i>True if the message references an Operation in the receiver</i>			

4.2.2 RoseRT

This section presents the RoseRT elements that are used in this case study. An overview of RoseRT was provided in Section 2.4.4. Diagrams of the metamodel can be found in Appendix C.

4.2.2.1 Attribute

Element: Attribute			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
type		String	
<i>The type of the attribute.</i>			

4.2.2.2 Parameter

Element: Parameter			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
type		String	
<i>The type of the Parameter.</i>			

4.2.2.3 Operation

Element: Operation			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
returnType		String	
<i>The type of the value returned by the operation.</i>			
parameters		Parameter	n
<i>A possibly empty set of parameters for the operation. This does not include the return parameter.</i>			

4.2.2.4 Class

Element: Class			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
attributes	Classifier	Attribute	n
<i>A possibly empty set of attributes that define the properties of the class.</i>			
operations	Classifier	Operation	n
<i>A possibly empty set of operations that define the capabilities of the class.</i>			
generalizations	Classifier	Generalization	n
<i>A possibly empty set of Generalization relationships owned by the Class. Defines the set of Classes that this Class specializes.</i>			

4.2.2.5 Capsule

Element: Capsule			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
attributes	Classifier	Attribute	n
<i>A possibly empty set of attributes that define the properties of the class.</i>			
operations	Classifier	Operation	n
<i>A possibly empty set of operations that define the capabilities of the class.</i>			
generalizations	Classifier	Generalization	n
<i>A possibly empty set of Generalization relationships owned by the Class. Defines the Capsule that this Capsule specializes.</i>			
structure		CapsuleStructure	1
<i>A mandatory definition of the structure of the Capsule</i>			

4.2.2.6 LogicalPackage

Element: LogicalPackage			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
logicalPackages		LogicalPackage	n
<i>A possibly empty set of LogicalPackages contained within this LogicalPackage</i>			
classes		Class	n
<i>A possibly empty set of Classes contained within this LogicalPackage</i>			
capsules		Capsule	n
<i>A possibly empty set of Capsules contained within this LogicalPackage</i>			
protocols		Protocol	n
<i>A possibly empty set of Protocols contained within this LogicalPackage</i>			
associations		Association	n
<i>A possibly empty set of Associations contained within this LogicalPackage</i>			

4.2.2.7 *AssociationEnd*

Element: AssociationEnd			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
aggregate		Boolean	
<i>Specifies whether this end of the Association is an aggregation with respect to the other end of the Association</i>			
containment		String	
<i>Specifies the containment of the Classifier at this end of the Association with respect to the other end of the Association. Possible values are "ByVal", "Ref" and "Unspecified"</i>			
multiplicity		String	
<i>Specifies the number of instances of the Classifier at this end of the Association that are associated with the Classifier at the other end of the Association</i>			
classifier		Classifier	1
<i>A mandatory property that specifies the Classifier (Class or Capsule) participating in the Association at this end of the Association</i>			

4.2.2.8 *Association*

Element: Association			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
end1		AssociationEnd	1
<i>Specifies the participant at one end of the Association relationship</i>			
end2		AssociationEnd	1
<i>Specifies the participant at the other end of the Association relationship</i>			

4.2.2.9 *Generalization*

Element: Generalization			
Feature	Inherited From	Type	Multiplicity
supplierClassifier	ClassRelation	Classifier	1
<i>The Class, Capsule or Protocol that the owning Class, Capsule or Protocol specializes respectively</i>			

4.2.2.10 Protocol

Element: Protocol			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
generalizations	Classifier	Generalization	n
<i>A possibly empty set of Generalization relationships owned by the Protocol. Defines the set of Protocols that this Protocol specializes.</i>			
inSignals		Signal	n
<i>A possibly empty set of Signals that the Protocol defines as incoming signals</i>			
outSignals		Signal	n
<i>A possibly empty set of Signals that the Protocol defines as outgoing signals</i>			

4.2.2.11 CapsuleStructure

Element: CapsuleStructure			
Feature	Inherited From	Type	Multiplicity
classifierRoles	Collaboration	ClassifierRole	n
<i>A possibly empty set of CapsuleRoles that form the internal structure of the Capsule that owns this CapsuleStructure</i>			
connectors	Collaboration	Connector	n
<i>A possibly empty set of Connectors that capture the communication relationships between the CapsuleRoles</i>			
ports		Port	n
<i>A possibly empty set of Ports that define the interface of the Capsule</i>			

4.2.2.12 Connector

Element: Connector			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
port1		Port	0..1
<i>The Port at the first end of the Connector when part of a CapsuleStructure</i>			
port2		Port	0..1
<i>The Port at the second end of the Connector when part of a CapsuleStructure</i>			

4.2.2.13 CapsuleRole

Element: CapsuleRole			
Feature	Inherited From	Type	Multiplicity
name		String	
capsule		Capsule	1
<i>The Capsule that this CapsuleRole is a projection of (See Section 2.4.4.2).</i>			

4.2.2.14 Port

Element: Port			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
conjugated		Boolean	
<i>Specifies whether the Port is conjugated, if it is conjugated the Protocol's in and out signals are reversed</i>			
protocol		Protocol	1
<i>The Protocol that this Port supports</i>			

4.2.2.15 Signal

Element: Signal			
Feature	Inherited From	Type	Multiplicity
name	Element	String	
type		String	
<i>The type of data that is sent/received with the Signal</i>			

4.3 Mapping Definitions

Mapping one modeling language into another modeling language is not an exact science. That is, many decisions are taken during the definition of the mappings that affect the result of executing the mappings. More specifically, two software engineers may define a different set of mappings when going from language A to language B and both sets of mappings may be considered acceptable. The acceptability of the mappings is context dependent and relies on either the software engineer having an in depth understanding of the domain and/or context, or the approval of an expert in the domain.

In this section we define *a* set of mappings for transforming a UML 1.5 model to a RoseRT model. These mappings do not consider all of the UML 1.5 metamodel concepts only the subset (see Section 4.2.1) that is used in this case study. We acknowledge that there may be another set of mappings from UML 1.5 to RoseRT that is applicable in a different domain and/or context.

The subset of UML 1.5 and RoseRT languages considered in this case study contain many similarities but it is the differences that make the mappings interesting. The total or two-way

encapsulation of capsules is the major difference between the two languages. This difference adds the following additional elements to RoseRT that do not have an equivalent semantic element in UML 1.5: CapsuleStructure, Port and Protocol. These additional elements are derived from UML 1.5 elements based on the context in which the relevant UML 1.5 elements are used. That is, a particular UML 1.5 element may map into different RoseRT elements depending upon the context in which it is used. For example, as we will see later in this section UML 1.5 Association elements can either map into a RoseRT Association or a RoseRT Protocol. The remainder of this section describes the overall mapping approach defined in the mappings.

In our mapping definitions, we consider a Package element to be the root element of a UML 1.5 model. A UML 1.5 Package is mapped into a RoseRT LogicalPackage, a package's contents are mapped based on their type. A UML 1.5 Class element is either mapped into a RoseRT Capsule or Class dependent upon whether it has been tagged as *active*. The tagging of a UML 1.5 Class as active is done through the additional information model. A set of strings called '*active*' is added to the additional information model. The '*active*' set contains the names of the classes that have been tagged as active. A UML 1.5 Interface is mapped into a RoseRT Protocol, as are UML 1.5 Associations between active classes. Those UML 1.5 Associations that are not between active classes are mapped into RoseRT Associations. Table 9 summarizes the mapping of a UML 1.5 Package element and its contents.

Table 9 Relationship between UML 1.5 Package, Class, Interface and Association and RoseRT LogicalPackage, Class, Capsule, Protocol and Association

UML 1.5 Element	RoseRT Element	Conditions
Package	LogicalPackage	
Class	Class	Class is not tagged as active
Class	Capsule	Class is tagged as active
Interface	Protocol	
Association	Association	At least one Class element involved in the Association is not active
Association	Protocol	The Class element at both ends of the Association is active

The sequence diagrams or interactions are used in the mappings to create signals in the Protocol elements that have been created from Association objects. More specifically, in the context of this case study the sequence diagrams define communication between active classes. The messages

between two active classes define a Signal element that is necessary for the active class to communicate. Since a signal is associated with an Operation element in the receiving Class we are able to create a Signal element for that message. In fact, two Signal elements are created one for sending the message and one for the result or return of the message²⁴. These signals are inserted into the Protocol that was created for the Association that exists between the two active classes. If no Association element exists between the two active classes then the Signal is dropped from the target model.

For example, suppose we have the UML 1.5 Class model shown in Figure 70: The Association element between the Client and ClientTransactionServer transforms into the Client_To_ClientTransactionServer protocol as shown in Figure 70. If a sequence diagram exists that shows an interaction pattern between Client and ClientTransactionServer we are able to generate signals for the Client_To_ClientTransactionServer (see, for example Figure 71). A signal in RoseRT can only have a single parameter; therefore a Class is created to wrap the parameters of an Operation. The class contains one Attribute for every parameter in the Operation.

---

²⁴ This is our resolution to the difference in inter-object communication paradigms used by the two languages. There may be other resolutions to this difference.

Figure 70 Mapping an active Association²⁵

Figure 71 Mapping a Message

As previously mentioned, Interfaces are mapped into Protocols. When an active class specializes an Interface, we create a Port in the corresponding capsule that implements the Protocol that was created for the Interface. For example, see Figure 72.

²⁵ The use of a comment to tag a Class is active is only for visualization. The actual tagging of a Class as discussed is realized through the additional information model.

Figure 72 Mapping a Generalization between a Interface and an active Class

Greater detail and the specifics of our UML 1.5 to RoseRT mappings can be found in the subsections that follow. For each mapping a short explanation is provided as well as the QVT mapping definition. Figure 73 shows a dependency graph for the mappings that have been defined. A directed edge from one mapping (node) to another mapping (node) states that the mapping at the tail (non arrowhead) uses or invokes the mapping at the head (arrowhead).

In the prototype the *UMLToRoseRT* mapping is the mapping start point. All of the other mappings are contained within the Mappings section of the compliance verification project that has been created for the case study.

Figure 73 UML 1.5 to RoseRT mapping dependency graph

4.3.1 AssociationEndsToPortName

Inputs	
<i>end1</i>	A UML AssociationEnd element that is connected to the Class that owns the Port.
<i>end2</i>	A UML AssociationEnd element that is connected to the Class at the other end of the Association from the Class that owns the Port.
Output	
A string value that is the name of the Port. The name is constructed by concatenating To_, the name of the Class at end2, _AS_ and the name of end1.	
Dependent on	
∅	
Rules dependent on it	
4.3.5 AssociationToPort	

4.3.1.1 QVT Mapping Definition

```

mapping AssociationEndsToPortName(end1, end2)
{
  domain {(AssociationEnd)[participant = p1, name = n1]}
  domain {(AssociationEnd)[participant = p2, name = n2]}
  body
  {
    "To_" + p2.name + "_AS_" + n1
  }
}

```

4.3.2 AssociationEndsToProtocolName

Inputs	
<i>end1</i>	A UML AssociationEnd element to be used in creating the protocol name.
<i>end2</i>	The other UML AssociationEnd element to be used in creating the protocol name.
Output	
A string value that is the name of the Protocol. The name is constructed by concatenating the name of end1, _TO_ and the name of end2.	
Dependent on	
∅	
Rules dependent on it	
4.3.6 AssociationToProtocol	

4.3.2.1 QVT Mapping Definition

```

mapping AssociationEndsToProtocolName(end1, end2)
{
  domain {(AssociationEnd)[participant = p1]}
  domain {(AssociationEnd)[participant = p2]}
  body
  {
    p1.name + "_TO_" + p2.name
  }
}

```

4.3.3 AssociationEndToAssociationEnd

Inputs	
<i>assocEnd</i>	The UML AssociationEnd element to be mapped into a RoseRT AssociationEnd.
Output	
A RoseRT AssociationEnd element, where the name, navigability and multiplicity are copied directly from the source element. The aggregate and containment fields are derived from the source element's aggregation field.	
Dependent on	
∅	
Rules dependent on it	
4.3.4 AssociationToAssociation	

4.3.3.1 QVT Mapping Definition

```

mapping AssociationEndToAssociationEnd(assocEnd)
{
  domain
  {
    (AssociationEnd)
    [
      name = n,
      isNavigable = nav,
      multiplicity = mult,
      aggregation = agg,
      participant = (Class)[objId = id, name = pn]
    ]
  }
  body
  {
    let
    cont = if agg = "aggregate" then "ByRef" else "Unspecified" endif
    cont = if agg = "composite" then "ByVal" else cont endif
    agt = if agg = "aggregate" or agg = "composite" then true else false endif
    in
    (AssociationEnd)
    [

```

```

    name = n,
    navigable = nav,
    aggregate = agt,
    containment = cont,
    multiplicity = mult,
    classifier = (Classifier) [objId = id, name = pn]
  ]
}
}

```

4.3.4 AssociationToAssociation

Inputs	
<i>assoc</i>	The UML Association element to be mapped into a RoseRT Association element.
Output	
A RoseRT Association element, whose name is copied from the source element. The ends of the Association are mapped from the source elements' ends.	
Dependent on	
4.3.3 AssociationEndToAssociationEnd	
Rules dependent on it	
$\emptyset$	

4.3.4.1 QVT Mapping Definition

```

mapping AssociationToAssociation(assoc)
{
  domain {(Association) [name = n, connection = ends]}
  body
  {
    let
      end1 = ends->at(1)
      end2 = ends->at(2)
    in
      (Association)
      [
        name = n,
        end1 = AssociationEndToAssociationEnd(end1),
        end2 = AssociationEndToAssociationEnd(end2)
      ]
  }
}

```

4.3.5 AssociationToPort

Inputs	
<i>association</i>	The UML Association element being mapped into a RoseRT Port element.
<i>endNo</i>	The end of the Association that is being mapped into a RoseRT Port element. The value expected is either a 1 or a 2.
Output	
A RoseRT Port element, whose name is based on the role it plays with respect to the Class at the other end of the Association.	
Dependent on	
4.3.1 AssociationEndsToPortName	
Rules dependent on it	
4.3.8 ClassToCapsule	

4.3.5.1 QVT Mapping Definition

```

mapping AssociationToPort(association, endNo)
{
  domain {(Association)[connection = memberEnds]}
  body
  {
    let
      end1 = memberEnds->at(1)
      end2 = memberEnds->at(2)
      pname =
        if endNo = 1 then
          AssociationEndsToPortName(end1, end2)
        else
          AssociationEndsToPortName(end2, end1)
        endif
      protocolName = end1.participant.name + "_TO_" + end2.participant.name
      role =
        if endNo = 1 then
          false
        else
          true
        endif
    in
      (Port)
      [
        name = pname,
        protocol = protocolName,
        conjugated = role
      ]
  }
}

```

4.3.6 AssociationToProtocol

Inputs	
<i>association</i>	The UML Association element that is being mapped into a RoseRT Protocol element.
Output	
A RoseRT Protocol element, whose name is based on the name of the participants at the ends of the Association.	
Dependent on	
4.3.2 AssociationEndsToProtocolName	
Rules dependent on it	
4.3.24 PackageToLogicalPackage	

4.3.6.1 QVT Mapping Definition

```

mapping AssociationToProtocol(association)
{
  domain {(Association)[connection = memberEnds]}
  body
  {
    let
      end1 = memberEnds->at(1)
      end2 = memberEnds->at(2)
      pname = AssociationEndsToProtocolName(end1, end2)
    in
      (Protocol)
      [
        name = pname,
        inSignals = Set {},
        outSignals = Set {}
      ]
  }
}

```

4.3.7 AttributeToAttribute

Inputs	
<i>attribute</i>	A UML Attribute element to be mapped into a RoseRT Attribute element.
Output	
A RoseRT Attribute element whose name and type fields are the same as the name and type features of the UML Attribute respectively.	
Dependent on	
∅	
Rules dependent on it	
4.3.9 ClassToDataClass	
4.3.8 ClassToCapsule	

4.3.7.1 QVT Mapping Definition

```

mapping AttributeToAttribute(attribute)
{
  domain { (Attribute) [ name = n, type = t] }
  body
  {
    (Attribute) [name = n, type = t]
  }
}

```

4.3.8 ClassToCapsule

Inputs	
<i>cls</i>	A UML Class element to be mapped into a RoseRT Capsule element.
<i>associations</i>	A set of UML Association elements that are used to create the Ports of the Capsule.
Output	
A RoseRT Capsule element whose name is the same as the Class that it was based on and whose attributes and operations are mapped from the source Class attributes and operations. The structure of the Capsule (i.e. Ports) is created using the associations set passed into the mapping.	
Dependent on	
4.3.7 AttributeToAttribute 4.3.22 OperationToOperation 4.3.5 AssociationToPort 4.3.12 GeneralizationToPort	
Rules dependent on it	
4.3.24 PackageToLogicalPackage	

4.3.8.1 QVT Mapping Definition

```

mapping ClassToCapsule(cls, associations)
{
  domain
  {
    (Class)
    [
      name = n,
      feature = ftrs,
      objId = id,
      generalization = gens
    ]
  }
  body
  {
    let
    portAssocEnd1 = associations->select(a |
      a.connection->at(1).participant.name = n)
  }
}

```

```

portAssocEnd2 = associations->select(a |
  a.connection->at(2).participant.name = n)
portEnd1 = portAssocEnd1->collect(a | AssociationToPort(a, 1))
portEnd2 = portAssocEnd2->collect(a | AssociationToPort(a, 2))
interfacePort =
  gens->select(g | g.parent.IsKindOf("Interface"))
  ->collect(i | GeneralizationToPort(i))
in
(Capsule)
[
  name = n,
  objId = id,
  attributes = ftrs->select(f |
    f.IsKindOf("Attribute"))->collect(a | AttributeToAttribute(a)),
  operations = ftrs->select(f |
    f.IsKindOf("Operation"))->collect(o | OperationToOperation(o)),
  structure =
    (CapsuleStructure)
    [
      ports = portEnd1->including(portEnd2)->including(interfacePort)->flatten()
    ]
]
}
}
}

```

4.3.9 ClassToDataClass

Inputs	
<i>cls</i>	A UML Class element to be mapped into a RoseRT Class element.
Output	
A RoseRT Class element whose name is the same as the Class that it was based on and whose attributes, operations and generalizations are mapped from the source Class attributes, operations and generalizations.	
Dependent on	
4.3.7 AttributeToAttribute 4.3.22 OperationToOperation 4.3.11 GeneralizationToGeneralization	
Rules dependent on it	
4.3.24 PackageToLogicalPackage	

4.3.9.1 QVT Mapping Definition

```

mapping ClassToDataClass(cls)
{
  domain
  {
    (Class)
    [
      name = n,
      feature = ftrs,

```

```

        objId = id,
        generalization = gens
    ]
}
body
{
    (Class)
    [
        name = n,
        objId = id,
        attributes = ftrs->select(f |
            f.IsKindOf("Attribute"))->collect( a | AttributeToAttribute(a) ),
        operations = ftrs->select(f |
            f.IsKindOf("Operation"))->collect( o | OperationToOperation(o) ),
        generalizations = gens->collect( g | GeneralizationToGeneralization(g) )
    ]
}
}
}

```

4.3.10 CollectProtocols

Inputs	
<i>lpackage</i>	The root package (LogicalPackage) from the new mapped RoseRT model.
Output	
A set containing all of the Protocols that have been created for the RoseRT tSpec model.	
Dependent on	
$\emptyset$	
Rules dependent on it	
4.3.28 UMLToRoseRT	

4.3.10.1 QVT Mapping Definition

```

mapping CollectProtocols(lpackage)
{
    domain
    {
        (LogicalPackage) [ protocols = p, logicalPackages = lpkgs ]
    }
    body
    {
        p->including( lpkgs->collect(pkg | CollectProtocols(pkg)))->flatten()
    }
}

```

4.3.11 GeneralizationToGeneralization

Inputs	
<i>umlgen</i>	A UML Generalization element that is to be mapped into a RoseRT Generalization element.
Output	
A RoseRT Generalization element whose supplierClassifier feature is based on the source elements parent feature.	
Dependent on	
Ø	
Rules dependent on it	
4.3.9 ClassToDataClass	

4.3.11.1 QVT Mapping Definition

```

mapping GeneralizationToGeneralization(umlgen)
{
  domain {(Generalization) [parent = p]}
  body
  {
    (Generalization)
    [
      supplierClassifier = (Classifier) [objId = p.objId, name = p.name]
    ]
  }
}

```

4.3.12 GeneralizationToPort

Inputs	
<i>generalization</i>	The UML Generalization element that is to be mapped into a RoseRT Port element.
Output	
A RoseRT Port element whose name is based on the specialized Interface element, and Protocol is the name of the specialized Interface element.	
Dependent on	
4.3.17 InterfaceToPortName	
Rules dependent on it	
4.3.8 ClassToCapsule	

4.3.12.1 QVT Mapping Definition

```

mapping GeneralizationToPort(generalization)
{
  domain {(Generalization) [parent = p]}
  body

```

```

{
  (Port)
  {
    name = InterfaceToPortName(p),
    conjugated = false,
    protocol = p.name
  }
}

```

4.3.13 InteractionAddToProtocol

Inputs	
<i>interaction</i>	A UML Interaction element whose messages are to be added to the tSpec Protocol elements.
<i>protocols</i>	The set of Protocol elements in the tSpec model.
Output	
There is no output, this mapping modifies the Protocol elements in the protocols set. It only adds those messages that are directly related to an operation in the Class that receives the message.	
Dependent on	
4.3.19 MessageToSignal	
Rules dependent on it	
4.3.27 ProcessInteractions	

4.3.13.1 QVT Mapping Definition

```

mapping InteractionAddToProtocol(interaction, protocols)
{
  domain {(Interaction) [messages = msgs]}
  body
  {
    let
      operationMsgs = msgs->select(m | m.isOperation = true)
    in
      operationMsgs->iterate(m | MessageToSignal(m, protocols))
  }
}

```

4.3.14 InteractionToDataClasses

Inputs	
<i>interaction</i>	The UML Interaction element to be mapped into a set of RoseRT Class elements.
Output	
A set of RoseRT Class elements, each Class is mapped from the Operation elements that are related to the messages in the Interaction.	
Dependent on	
4.3.21 OperationToDataClass	
Rules dependent on it	
4.3.24 PackageToLogicalPackage	

4.3.14.1 QVT Mapping Definition

```

mapping InteractionToDataClasses(interaction)
{
  domain {(Interaction)[messages = msgs]}
  body
  {
    let
      operationMsgs = msgs->select(m | m.isOperation = true)
    in
    {
      operationMsgs->select(m |
        m.operation.parameter->select(p | p.kind != "return")->notEmpty())
        ->collect(m | OperationToDataClass(m.operation))
    }
  }
}

```

4.3.15 InterfaceOperationToSignal

Inputs	
<i>operation</i>	The UML Operation element to be mapped into a RoseRT Signal element.
Output	
A RoseRT Signal element whose name is the same as the source Operation and whose type is based on the parameters of the source Operation.	
Dependent on	
∅	
Rules dependent on it	
4.3.18 InterfaceToProtocol	
Notes	
This mapping is only to be invoked by the InterfaceToProtocol mapping.	

4.3.15.1 QVT Mapping Definition

```

mapping InterfaceOperationToSignal(operation, interfaceName)
{
  domain {(Operation) [name = n, parameter = prms]}
  body
  {
    let
      stype =
        if prms->select(p | p.kind != "return")->size() = 0 then
          "void"
        else
          n + "DATA_CLASS_FROM" + interfaceName
        endif
    in
      (Signal)
      [
        name = n,
        type = stype
      ]
  }
}

```

4.3.16 InterfaceToDataClasses

Inputs	
<i>interface</i>	A UML Interface element to be mapped into a set of RoseRT Class elements.
Output	
A set of RoseRT Class elements that are derived from the parameters of the source Interface's operations.	
Dependent on	
4.3.21 OperationToDataClass	
Rules dependent on it	
4.3.24 PackageToLogicalPackage	

4.3.16.1 QVT Mapping Definition

```

mapping InterfaceToDataClasses(interface)
{
  domain{(Interface)[name = n, feature = ftr]}
  body
  {
    ftr->select(f | f.IsKindOf("Operation"))->select(oper |
      oper.parameter->select(param | param.kind != "return")->size() != 0)
      ->collect( op | OperationToDataClass(op, n))
  }
}

```

4.3.17 InterfaceToPortName

Inputs	
<i>interface</i>	A UML Interface element that is to be mapped into a string.
Output	
A string value that is the name of a RoseRT Port element. The value is created by concatenating to_ with the name of the specialized Interface element.	
Dependent on	
Ø	
Rules dependent on it	
4.3.12 GeneralizationToPort	

4.3.17.1 QVT Mapping Definition

```

mapping InterfaceToPortName(interface)
{
  domain {(Interface) [name = n]}
}

```



```

body
{
  "to_" + n
}
}

```

4.3.18 InterfaceToProtocol

Inputs	
<i>interface</i>	A UML Interface element that is to be mapped into a RoseRT Protocol element.
Output	
A RoseRT Protocol element, whose name is the same as the source Interface element and whose inSignals are derived from mapping the operations of the source Interface and whose outSignals are derived from mapping the return parameters of the source Interface's operations.	
Dependent on	
4.3.15 InterfaceOperationToSignal 4.3.20 OperationResultToSignal	
Rules dependent on it	
4.3.24 PackageToLogicalPackage	

4.3.18.1 QVT Mapping Definition

```

mapping InterfaceToProtocol(interface)
{
  domain {(Interface)[name = n, feature = ftr]}
  body
  {
    (Protocol)
    [
      name = n,
      inSignals = ftr->select( f | f.IsKindOf("Operation"))
        ->collect( op | InterfaceOperationToSignal(op)),
      outSignals = ftr->select( f | f.IsKindOf("Operation"))
        ->collect( op | OperationResultToSignal(op, n))
    ]
  }
}

```

4.3.19 MessageToSignal

Inputs	
<i>message</i>	The UML Message element to be mapped into a RoseRTSignal.
<i>protocols</i>	The Protocols in the tSpec model.
Output	
There is no output to this mapping: it creates two RoseRT Signal elements from the source Message element and adds them to the inSignals and outSignals of a Protocol found in protocols.	
Dependent on	
4.3.23 OperationToSignal 4.3.20 OperationResultToSignal	
Rules dependent on it	
4.3.13 InteractionAddToProtocol	

4.3.19.1 QVT Mapping Definition

```

mapping MessageToSignal(message, protocols)
{
  domain {(Message)[sender = s, receiver = r, operation = op]}
  body
  {
    let
      senderBase =
        if protocols->select( prot | prot.name = s + "_TO_" + r)->isEmpty() then
          false
        else
          true
        endif
      p =
        if protocols->select( prot | prot.name = s + "_TO_" + r)->isEmpty() then
          protocols->select( prot | prot.name = r + "_TO_" + s)->first()
        else
          protocols->select( prot | prot.name = s + "_TO_" + r)->first()
        endif
      signal = OperationToSignal(op)
      rsignal = OperationResultToSignal(op)
    in
      if senderBase then
        {
          if p.outSignals->exists(s | s.name = signal.name) then
            ""
          else
            {
              p.outSignals := p.outSignals->including(signal);
              p.inSignals := p.inSignals->including(rsignal)
            }
          }
        }
  }
}

```

```

        endif
    }
else
{
    if p.outSignals->exists(s | s.name = rsignal.name) then
        ""
    else
    {
        p.outSignals := p.outSignals->including(rsignal);
        p.inSignals := p.inSignals->including(signal)
    }
    endif
}
endif
}
}
}

```

4.3.20 OperationResultToSignal

Inputs	
<i>operation</i>	A UML Operation element that is to be mapped into a RoseRT Signal.
Output	
A RoseRT Signal whose name is the name of the source Operation with <code>_RESULT</code> appended to it and the type of the Signal is the return type of the source Operation.	
Dependent on	
$\emptyset$	
Rules dependent on it	
4.3.19 MessageToSignal	

4.3.20.1 QVT Mapping Definition

```

mapping OperationResultToSignal(operation)
{
    domain {(Operation)[name = n, parameter = prms]}
    body
    {
        let
            retParam =
                if prms->select(p | p.kind = "return")->isEmpty() then
                    ""
                else
                    prms->select(p | p.kind = "return")->first()
                endif
            stype =
                if retParam = "" then
                    "void"
                else
                    retParam.type
                endif
        endlet
    }
}

```

```

    in
    (Signal)
    {
        name = n + "_RESULT",
        type = stype
    }
}

```

4.3.21 OperationToDataClass

Inputs	
<i>operation</i>	A UML Operation element that is to be mapped into a RoseRT Class element.
Output	
A RoseRT Class element whose name is the source Operation's name with DATA_CLASS_FROM appended to it and whose attribute set is mapped from the non-return Parameters of the source Operation. The operations and generalizations features are empty sets.	
Dependent on	
4.3.25 ParameterToAttribute	
Rules dependent on it	
4.3.16 InterfaceToDataClasses	
4.3.14 InteractionToDataClasses	

4.3.21.1 QVT Mapping Definition

```

mapping OperationToDataClass(operation, classifierName)
{
    domain
    {
        (Operation)
        [
            name = n,
            parameter = prms
        ]
    }
    body
    {
        (Class)
        [
            name = n + "DATA_CLASS_FROM" + classifierName,
            attributes =
                prms->select(p |
                    p.kind != "return")->collect( param | ParameterToAttribute(param)),
            operations = Set{},
            generalizations = Set{}
        ]
    }
}

```

4.3.22 OperationToOperation

Inputs	
<i>operation</i>	A UML Operation element that is to be mapped into a RoseRT Operation element.
Output	
A RoseRT Operation whose name is the same of the source Operation element, returnType is the type of the source Operation's return Parameter and parameters are mapped from the source Operation's parameters except for the return parameter.	
Dependent on	
4.3.26 ParameterToParameter	
Rules dependent on it	
4.3.9 ClassToDataClass 4.3.8 ClassToCapsule	

4.3.22.1 QVT Mapping Definition

```

mapping OperationToOperation(operation)
{
  domain { (Operation) [name = n, parameter = parms] }
  body
  {
    let
      opParamSet = parms->select(p1 | p1.kind != "return")
      retParamSet = parms->select(p2 | p2.kind = "return")
      retParam =
        if retParamSet->isEmpty() then
          "null"
        else
          retParamSet->first()
        endif
    in
      (Operation)
      [
        name = n,
        parameters = opParamSet->collect( p3 | ParameterToParameter(p3) ),
        returnType = if retParam = "null" then "" else retParam.type endif
      ]
  }
}

```

4.3.23 OperationToSignal

Inputs	
<i>operation</i>	A UML Operation element that is to be mapped into a RoseRT Signal element.
Output	
A RoseRT Signal element whose name is the same as the source Operation's name and type is the empty string if there are no parameters otherwise it is the source Operation name with DATA_CLASS_FROM appended to it.	
Dependent on	
$\emptyset$	
Rules dependent on it	
4.3.19 MessageToSignal	

4.3.23.1 QVT Mapping Definition

```

mapping OperationToSignal(operation, classifierName)
{
  domain {(Operation)[name = n, parameter = prms]}
  body
  {
    let
      stype =
        if prms->select(p | p.kind != "return")->size() = 0 then
          "void"
        else
          n + "DATA_CLASS_FROM" + classifierName
        endif
    in
      (Signal)
      [
        name = n,
        type = stype
      ]
  }
}

```

4.3.24 PackageToLogicalPackage

Inputs	
<i>pkg</i>	A UML Package element that is to be mapped into a RoseRT LogicalPackage element.
<i>additional</i>	The additional information used in mapping the source Package element.
Output	
A RoseRT LogicalPackage element whose name is the name of the source Package element and whose ▪ logicalPackages are derived from the sources Package elements; ▪ classes are derived from the sources non active classes, interface operations, and interaction messages; ▪ capsules are derived from the sources active classes; ▪ associations are derived from the sources associations between non-active classes; and ▪ protocols are derived from the source associations between active classes and source interfaces	
Dependent on	
4.3.24 PackageToLogicalPackage 4.3.9 ClassToDataClass 4.3.8 ClassToCapsule 4.3.16 InterfaceToDataClasses 4.3.6 AssociationToProtocol 4.3.18 InterfaceToProtocol 4.3.14 InteractionToDataClasses 4.3.4 AssociationToAssociation	
Rules dependent on it	
4.3.28 UMLToRoseRT	

4.3.24.1 QVT Mapping Definition

```

mapping PackageToLogicalPackage(pkg, additional)
{
  domain { (Package) [name = n, ownedElement = els] }
  domain { (AdditionalInfo, ai) [active = actCls] }
  body
  {
    let
      mappedPackages = els->select(cls | cls.IsKindOf("Package"))
      ->collect(p | PackageToLogicalPackage(p, ai))
      mappedClasses = els
      ->select(cls | cls.IsKindOf("Class") and actCls->excludes(cls.name))
      ->collect(c | ClassToDataClass(c))
      interfaceClasses = els->select(cls | cls.IsKindOf("Interface"))
      ->collect(inter | InterfaceToDataClasses(inter))->flatten()
  }
}

```

```

activeAssociations = els->select(cls | cls.IsKindOf("Association"))
  ->select(a | actCls->includes(a.connection->at(1).participant.name)
    and actCls->includes(a.connection->at(2).participant.name))
nonActiveAssociations = els->select(cls | cls.IsKindOf("Association"))
  ->select(a | actCls->excludes(a.connection->at(1).participant.name)
    or actCls->excludes(a.connection->at(2).participant.name))
associationProtocols = activeAssociations->collect(a | AssociationToProtocol(a))
interfaceProtocols = els->select(cls | cls.IsKindOf("Interface"))
  ->collect(inter | InterfaceToProtocol(inter))
interactionClasses = els->select(cls | cls.IsKindOf("Interaction"))
  ->collect(i | InteractionToDataClasses(i, associationProtocols))
  ->unique()
in
(LogicalPackage)
[
  name = n,
  logicalPackages = mappedPackages,
  classes = mappedClasses->including(interfaceClasses)
    ->including(interactionClasses)->flatten(),
  capsules = els->select(cls | cls.IsKindOf("Class"))
    and actCls->includes(cls.name))
    ->collect(c | ClassToCapsule(c, activeAssociations)),
  protocols = interfaceProtocols->including(associationProtocols)->flatten(),
  associations = nonActiveAssociations
    ->collect(assoc | AssociationToAssociation(assoc, mappedClasses, actCls))
]
}
}

```

4.3.25 ParameterToAttribute

Inputs	
<i>parameter</i>	A UML Parameter element that is to be mapped into a RoseRT Attribute element
Output	
A RoseRT Attribute whose name and type are the same as the source Parameters'.	
Dependent on	
Ø	
Rules dependent on it	
4.3.21 OperationToDataClass	

4.3.25.1 QVT Mapping Definition

```

mapping ParameterToAttribute(parameter)
{
  domain {(Parameter)[name = n, type = t]}
  body
  {
    (Attribute)[name = n, type = t]
  }
}

```


4.3.26 ParameterToParameter

Inputs	
<i>param</i>	A UML Parameter element that is to be mapped into a RoseRT Parameter element
Output	
A RoseRT Parameter whose name and type are the same as the source Parameters'.	
Dependent on	
Ø	
Rules dependent on it	
4.3.22 OperationToOperation	

4.3.26.1 QVT Mapping Definition

```

mapping ParameterToParameter(param)
{
  domain { (Parameter) [name = n, type = t] }
  body
  {
    (Parameter) [name = n, type = t]
  }
}

```

4.3.27 ProcessInteractions

Inputs	
<i>pkg</i>	The UML Package element whose Interaction elements are to be processed.
<i>protocols</i>	The set of Protocols contained in the tSpec model.
Output	
This mapping does not produce an output rather it processes the Interaction elements in a Package element and its sub-packages.	
Dependent on	
4.3.13 InteractionAddToProtocol	
Rules dependent on it	
4.3.28 UMLToRoseRT	

4.3.27.1 QVT Mapping Definition

```

mapping ProcessInteractions(pkg, protocols)
{
  domain {(Package)[ownedElement = els]}
  body
  {

```

```

{
  els->select(cls | cls.IsKindOf("Interaction"))
  ->collect(i | InteractionAddToProtocol(i, protocols));
  els->select(cls | cls.IsKindOf("Package"))
  ->collect(p | ProcessInteractions(p, protocols))
}
}
}

```

4.3.28 UMLToRoseRT

Inputs	
<i>package</i>	the root model package, which is a UML Package element
<i>additional</i>	the additional information used in the mapping process this is a AdditionalInfo element
Output	
A RoseRT LogicalPackage element that represents the root of the RoseRT tSpec model.	
Dependent on	
4.3.24 PackageToLogicalPackage 4.3.27 ProcessInteractions	
Rules dependent on it	
Ø	
Notes	
The UMLToRoseRT mapping is the starting point for mapping a UML specification model into a RoseRT tSpec model. The mapping assumes that the specification model is contained within a root package, because in the case of the Rational Rose tool it is (the LogicalPackage).	

4.3.28.1 QVT Mapping Definition

```

mapping UMLToRoseRT(package, additional)
{
  domain { (Package) [name = n, ownedElement = els] }
  domain { (AdditionalInfo, ai) [active = actCls] }
  body
  {
    let
      pkg = PackageToLogicalPackage(package, additional)
      allProtocols = CollectProtocols(pkg)
    in
    {
      ProcessInteractions(package, allProtocols);
      pkg
    }
  }
}

```

4.4 Mapping Verification

In this section we provide the test models for two of the mappings defined in the previous section. For the two mappings we provide the combinatorial model we have developed to verify the mapping. A project is created specifically for verifying each mapping. Recall that for verifying a particular mapping we first develop the combinatorial model from which a test model is built. The *specification* model for the project we have created is set to this test model. In order to execute the mapping under test (MUT) we define a test driver, which becomes the mapping starting point. Within the mappings section of the project the MUT is included along with its supporting mappings. The supporting mappings can be determined from the mapping dependency graph that is provided in Figure 73. In presenting the combinatorial models for mappings we have chosen a mapping whose input element contains a set (e.g. feature in a Interface element) and a second mapping whose input element has a larger number of attributes. The two mappings that the combinatorial model is provided for are *InterfaceToProtocol* and *AssociationEndToAssociationEnd*.

4.4.1 Verifying InterfaceToProtocol

The QVT mapping definition for this mapping is provided in Section 4.3.18. By analyzing the mapping we see that there are two fields or features of an Interface that affect the output of the mapping: *name* and *feature*. The *name* feature is a String that is copied to the *name* of the Protocol. We consider two choices for name: an empty-string and a non-empty string such as "ClientInterface".

The '*feature*' feature of an Interface is a set that can contain both Operation and Attribute and attribute elements. This particular mapping is only interested in those elements of the set that are of type Operation. Therefore, all non-Operation elements are considered equivalent. In testing the mapping we consider the following classes of sets: empty set, set containing only Operation elements, set containing only non-Operation elements and the set containing both Operation and non-Operation elements. The choices for the InterfaceToProtocol mapping are summarized in

Table 10. We note that there could be other possible choices: this is but one possible example of the choices.

Table 10 Choices for features that affect the InterfaceToProtocol mapping

	Choices	Notes
Object Type	1. Interface 2. not Interface	Defines the type of the input element. We consider anything other than Interface to be equivalent.
name	1. a name value (length ≥ 1) 2. empty-string	This is a string value and we are interested in when it is a string of at least one character and the special case when the string is empty.
feature	1. empty-set 2. set containing only Operation elements (singleton) 3. set containing only Operation elements (size > 1) 4. set containing only non-Operation elements (singleton) 5. set containing only non-Operation elements (size > 1) 6. set containing both Operation and non-Operation elements (size = 2) 7. set containing both Operation and non-Operation elements (size > 2)	The feature set can contain both Attribute and Operation elements.

Taking into account the choices defined in Table 10, we derive the combinatorial model shown in Table 11 for the *InterfaceToProtocol* mapping. An explanation for the symbols used in the model is provided in Table 12. We have also included a test (first row) to verify that the mapping fails if the input object is not an Interface.

Table 11 Combinatorial model for verifying the InterfaceToProtocol mapping

	Input			Expected Result			
	Object Type	name	feature	Object Type	name	inSignals	outSignals
1	! Interface	D/C	D/C	FAIL			
2	Interface	""	$\emptyset$	Protocol	""	$\emptyset$	$\emptyset$
3	Interface	η	$\emptyset$	Protocol	η	$\emptyset$	$\emptyset$
4	Interface	""	Ω_{op}	Protocol	""	Ω_{in}	Ω_{out}
5	Interface	""	Ω_{op}'	Protocol	""	Ω_{in}	Ω_{out}
6	Interface	η	Ω_{op}	Protocol	η	Ω_{in}	Ω_{out}
7	Interface	η	Ω_{op}'	Protocol	η	Ω_{in}	Ω_{out}
8	Interface	""	Ω_{nop}	Protocol	""	$\emptyset$	$\emptyset$
9	Interface	""	Ω_{nop}'	Protocol	""	$\emptyset$	$\emptyset$
10	Interface	η	Ω_{nop}	Protocol	η	$\emptyset$	$\emptyset$
11	Interface	η	Ω_{nop}'	Protocol	η	$\emptyset$	$\emptyset$
12	Interface	""	Ω_m	Protocol	""	Ω_{inm}	Ω_{outm}
13	Interface	""	Ω_m'	Protocol	""	Ω_{inm}	Ω_{outm}
14	Interface	η	Ω_m	Protocol	η	Ω_{inm}	Ω_{outm}
15	Interface	η	Ω_m'	Protocol	η	Ω_{inm}	Ω_{outm}

Table 12 Explanation of symbols used in Table 11

Symbol	Explanation
η	a string value that represents a name, it is held constant for a particular row
$\emptyset$	an empty set
Ω_{op}'	the set of features only contains Operation elements (singleton)
Ω_{op}	the set of features only contains Operation elements (size > 1)
Ω_{nop}'	the set of features only contains non-Operation elements (singleton)
Ω_{nop}	the set of features only contains non-Operation elements (size > 1)
Ω_m'	the set of features contains both Operation elements and non-Operation elements (singleton)
Ω_m	the set of features contains both Operation elements and non-Operation elements (size > 1)
Ω_{in}	a set of Signal elements, where there is one Signal element for each element contained in the Ω_{op} or Ω_{op}' set of its respective row, each Signal element is the result of applying the InterfaceOperationToSignal mapping on an Operation element
Ω_{out}	a set of Signal elements, where there is one Signal element for each element contained in the Ω_{op} or Ω_{op}' set of its respective row, each Signal element is the result of applying the OperationResultToSignal mapping on an Operation element
Ω_{inm}	a set of Signal elements, where there is one Signal element for each Operation element contained in the Ω_m or Ω_m' set of its respective row, each Signal element is the result of applying the InterfaceOperationToSignal mapping on an Operation element. Not all elements of Ω_m or Ω_m' are mapped.
Ω_{outm}	a set of Signal elements, where there is one Signal element for each Operation element contained in the Ω_m or Ω_m' set of its respective row, each Signal element is the result of applying the OperationResultToSignal mapping on an Operation element. Not all elements of Ω_m or Ω_m' are mapped.
D/C	the value doesn't matter, it is best to choose a valid value for that particular parameter

Following the creation of the combinatorial model a test driver is developed. The test driver is itself a QVT mapping that takes as input a test model and invokes the MUT on each of the tests within the test model. For example, a test driver for the InterfaceToProtocol mapping is provided in Figure 74. This test driver becomes the mapping starting point in the test project we have created. Examining the mapping dependency in Figure 73 reveals that both InterfaceOperationToSignal and OperationResultToSignal must also be included in the project as the InterfaceToProtocol mapping is dependent upon them.

```

mapping InterfaceToProtocolDriver(testModel)
{
  domain { (InterfaceTestModel)[ tests = t ]}
  body
  {
    (InterfaceTestResult)
    [
      results = t->collect(test |
        (TestCaseResult)
        [
          testcase = test.testcase,
          result = InterfaceToProtocol(test)
        ]
      )
    ]
  }
}

```

Figure 74 Test driver for testing the InterfaceToProtocol mapping

In verifying the InterfaceToProtocol mapping two instantiations of the combinatorial model were constructed. An instantiation of the combinatorial model is a test suite that contains a test case for each row in the model. That is, for each row in the combinatorial model an input object is created with the properties defined in the model. For example, Figure 75 defines an input object that corresponds to row 5 of Table 11. Each of these instantiations was used as input to the test driver shown in Figure 74 and a results model was produced. Analyzing the results model we found that the results were what was expected.

Figure 75 Example of an Interface element that corresponds to row 5 of Table 11

4.4.2 Verifying AssociationEndToAssociationEnd

The QVT mapping definition for the AssociationEndToAssociationEnd mapping is provided in Section 4.3.3. By analyzing the mapping we see that there are six (6) features or fields of an AssociationEnd element that affect the output of the mapping: object type, name, isNavigable, multiplicity, aggregation and participant. For each of these features we define a set of choices, which define the possible values for that particular feature. Table 13 shows each of the features with their choices. We note that this particular mapping does not contain a feature that is a set of values.

Table 13 AssociationEnd features and their choices

Parameter	Choices	Notes
Object Type	1. AssociationEnd 2. not AssociationEnd	Defines the type of the input object. We consider anything other than AssociationEnd to be equivalent.
name	1. a name value (length ≥ 1) 2. empty-string	The name of the AssociationEnd.
isNavigable	1. true 2. false	
multiplicity	1. a multiplicity value (length ≥ 1) 2. empty-string	
aggregation	1. aggregate 2. composite 3. empty-string	Possible values are: aggregate, composite and empty-string.
participant	1. a Class object 2. not a Class object	Expected to be a Class object. We consider anything other than a Class object to be equivalent.

The choices for each feature shown in Table 13 can be used to build a combinatorial model for testing the AssociationEndToAssociationEnd mapping. For the input section of the model, a row is defined for each valid combination of the choices. Once the input combinations have been defined, the expected value(s) for the input is determined. The expected value for a particular input is shown in the Expected Results section of a row. The target element is an AssociationEnd element with six features: name, navigable, aggregate, containment, multiplicity and classifier. The name, navigable and multiplicity features of the expected result should be a copy of the name, isNavigable and multiplicity input values respectively. Expected results fields; aggregate and containment; are dependent on the aggregation input value. Similarly, the classifier expected result feature is dependent on the participant input value.

The model shown in Table 14 captures the input combinations and their expected values that were discussed in the previous paragraph. The symbols used in the model are defined in Table 16. In verifying the AssociationEndToAssociationEnd mapping an instantiation of this combinatorial model was constructed. The instantiation contains an object for each row in the model. The object for a particular row has the properties defined in the input section of the row. For example, Figure 75 defines instantiation objects for rows 15 and 25 of Table 14. This instantiation was used as input to the test driver shown in Figure 77 and a results model was produced. Analyzing the results model we were able to determine that the results agreed with the expected results section of the combinatorial model.

Table 14 Combinatorial model defining test cases for verifying the AssociationEndToAssociationEnd mapping

	Input						Expected Result						
	Object Type	name	isNavigable	multiplicity	aggregation	participant	Object Type	name	navigable	aggregate	containment	multiplicity	classifier
1	AssociationEnd	D/C	D/C	D/C	D/C	D/C	Fail						
2	AssociationEnd	""	T	""	agg	ζ	AssociationEnd	""	T	T	Ref	""	ζ
3	AssociationEnd	D/C	D/C	D/C	D/C	!ζ	Fail						
4	AssociationEnd	""	T	""	comp	ζ	AssociationEnd	""	T	T	Val	""	ζ
5	AssociationEnd	""	T	""	""	ζ	AssociationEnd	""	T	F	Uns	""	ζ
6	AssociationEnd	""	T	M	agg	ζ	AssociationEnd	""	T	T	Ref	M	ζ
7	AssociationEnd	""	T	M	comp	ζ	AssociationEnd	""	T	T	Val	M	ζ
8	AssociationEnd	""	T	M	""	ζ	AssociationEnd	""	T	T	Uns	M	ζ
9	AssociationEnd	""	F	""	agg	ζ	AssociationEnd	""	F	T	Ref	""	ζ
10	AssociationEnd	""	F	""	comp	ζ	AssociationEnd	""	F	T	Val	""	ζ
11	AssociationEnd	""	F	""	""	ζ	AssociationEnd	""	F	F	Uns	""	ζ
12	AssociationEnd	""	F	M	agg	ζ	AssociationEnd	""	F	F	Ref	M	ζ
13	AssociationEnd	""	F	M	comp	ζ	AssociationEnd	""	F	T	Val	M	ζ
14	AssociationEnd	""	F	M	""	ζ	AssociationEnd	""	F	F	Uns	M	ζ
15	AssociationEnd	η	T	""	agg	ζ	AssociationEnd	η	T	T	Ref	""	ζ
16	AssociationEnd	η	T	""	comp	ζ	AssociationEnd	η	T	T	Val	""	ζ

Table 15 Combinatorial model defining test cases for verifying the AssociationEndToAssociationEnd mapping (continued)

	Input						Expected Result						
	Object Type	name	isNavigable	multiplicity	aggregation	participant	Object Type	name	navigable	aggregate	containment	multiplicity	classifier
17	AssociationEnd	η	T	""	""	ς	AssociationEnd	η	T	F	Uns	""	ς'
18	AssociationEnd	η	T	M	agg	ς	AssociationEnd	η	T	T	Ref	M	ς
19	AssociationEnd	η	T	M	comp	ς	AssociationEnd	η	T	T	Val	M	ς'
20	AssociationEnd	η	T	M	""	ς	AssociationEnd	η	T	F	Uns	M	ς
21	AssociationEnd	η	F	""	agg	ς	AssociationEnd	η	F	T	Ref	""	ς'
22	AssociationEnd	η	F	""	comp	ς	AssociationEnd	η	F	T	Val	""	ς
23	AssociationEnd	η	F	""	""	ς	AssociationEnd	η	F	F	Uns	""	ς'
24	AssociationEnd	η	F	M	agg	ς	AssociationEnd	η	F	T	Ref	M	ς
25	AssociationEnd	η	F	M	comp	ς	AssociationEnd	η	F	T	Val	M	ς'
26	AssociationEnd	η	F	M	""	ς	AssociationEnd	η	F	F	Uns	M	ς

Table 16 Explanation of symbols used in Table 14

Symbol	Explanation
η	a name value, which remains constant for a particular row
T	true
F	false
M	a multiplicity value, which remains constant for a particular row
agg	aggregate
comp	composite
Ref	ByRef
Val	ByVal
Uns	Unspecified
$!\varsigma$	not a Class object
ς	a Class object
ς'	a Classifier object that is related to the ς in its respective row by $\varsigma' = (\text{Classifier})[\text{objId} = \varsigma.\text{objId}, \text{name} = \varsigma.\text{name}]$

Figure 76 Instantiation of test cases 15 and 25 from Table 14

```

mapping AssociationEndToAssociationEndDriver(testModel)
{
  domain { (AssociationEndTestModel)[ tests = t ]}
  body
  {
    (AssociationEndTestResult)
    [
      results = t->collect(test |
        (TestCaseResult)
        [
          testcase = test.testcase,
          result = AssociationEndToAssociationEnd(test)
        ]
      )
    ]
  }
}

```

Figure 77 Test driver for the AssociationEndToAssociationEnd mapping

4.4.3 Verifying remaining mappings

In the previous two sections the combinatorial model and test driver have been presented for verifying two of the mappings. The development of the combinatorial model and test driver is a

repeatable process and therefore we do not feel that including the combinatorial model and test driver for the entire set of mappings contributes further to this dissertation.

Through the definition of the mappings in Section 4.3 and their verification we have found that this is an iterative process. Similar, to the development of software in a programming language (e.g., JAVA), the definition of the mappings were defined and verified in an incremental manner. That is, a mapping is defined and verified before it is used within another mapping. Furthermore, the complexity of a mapping definition is built up incrementally. The features of an input element are added to the mapping definition incrementally and mapping verified to ensure that the particular feature is correct. We observed this through our mapping definition process, where the initial mapping definition often contained errors. For example, in mapping a UML 1.5 AssociationEnd to a RoseRT AssociationEnd the relationship between ‘*aggregation*’ in UML 1.5 and ‘*aggregate*’ and ‘*containment*’ in RoseRT was incorrectly defined in the AssociationEndToAssociationEnd mapping.

4.5 Compliance Rules

Figure 78 RoseRT design to RoseRT tSpec compliance rule dependency graph

4.5.1 AssociationCompliance

This relation specifies the rules for compliance between two RoseRT Associations. It states that two Associations are compliant if they have the same name and their ends are compliant based on the compliance rule *AssociationEndCompliance* (4.5.2).

Dependent on	4.5.2 AssociationEndCompliance
Rules dependent on it	4.5.11 PackageCompliance

4.5.1.1 QVT Relation Definition

```

relation AssociationCompliance
{
  domain {(Association)[name = n, end1 = end1Spec, end2 = end2Spec]}
  domain { Association}[name = n, end1 = end1Design, end2 = end2Design] }
  when
  {
    AssociationEndCompliance(end1Spec, end1Design) and
    AssociationEndCompliance(end2Spec, end2Design)
  }
}

```

4.5.2 AssociationEndCompliance

The *AssociationEndCompliance* relation defines the rules for compliance between two RoseRT AssociationEnds. Two AssociationEnds are compliant if all of their features are equal. The rule is not dependent on any of the other compliance rules.

Dependent on	$\emptyset$
Rules dependent on it	4.5.1 AssociationCompliance

4.5.2.1 QVT Relation Definition

```

relation AssociationEndCompliance
{
  domain
  {
    (AssociationEnd)
    [ name = n, aggregate = agg, containment = cont, multiplicity = mult, classifier = cSpec ]
  }
  domain
  {
    (AssociationEnd)
    [ name = n, aggregate = agg, containment = cont, multiplicity = mult, classifier = cDesign ]
  }
}

```

```

when
{
  cSpec.name = cDesign.name
}
}

```

4.5.3 AttributeCompliance

The *AttributeCompliance* rule defines compliance between two RoseRT Attributes. A *design* Attribute is considered compliant with a *tSpec* Attribute if they have equal *name* and *type* features.

Dependent on	$\emptyset$
Rules dependent on it	4.5.4 CapsuleCompliance 4.5.7 ClassCompliance

4.5.3.1 QVT Relation Definition

```

relation AttributeCompliance
{
  domain {(Attribute) [name = n, type = t]}
  domain {(Attribute) [name = n, type = t]}
}

```

4.5.4 CapsuleCompliance

The CapsuleCompliance relation defines the compliance rules between a RoseRT *design* Capsule element and a RoseRT *tSpec* Capsule element. A *design* Capsule is considered compliant with the *tSpec* if its name is the same as the *tSpec* name, its attribute set contains at least one compliant Attribute for each of the attributes in the *tSpec* attribute set, the same between the two operation sets and generalization sets. That is, the *design* model may define additional attributes, operations and generalizations. Finally, the *design* Capsule's structure must be compliant with the *specification* structure. Attribute compliance is verified by the AttributeCompliance rule, Operation compliance by the OperationCompliance rule, Generalization compliance by the GeneralizationCompliance rule and structure compliance by the CapsuleStructureCompliance rule.

Dependent on	4.5.3 AttributeCompliance 4.5.10 OperationCompliance 4.5.9 GeneralizationCompliance 4.5.6 CapsuleStructureCompliance
Rules dependent on it	4.5.11 PackageCompliance

4.5.4.1 QVT Relation Definition

```

relation CapsuleCompliance
{
  domain
  {
    (Capsule)
    [ name = n, attributes = aSpec, operations = oSpec, generalizations = gSpec, structure = sSpec ]
  }
  domain
  {
    (Capsule)
    [
      name = n,
      attributes = aDesign,
      operations = oDesign,
      generalizations = gDesign,
      structure = sDesign
    ]
  }
  when
  {
    aSpec->forAll(a | AttributeCompliance(a, aDesign)) and
    oSpec->forAll(o | OperationCompliance(o, oDesign)) and
    gSpec->forAll(g | GeneralizationCompliance(g, gDesign)) and
    CapsuleStructureCompliance(sSpec, sDesign)
  }
}

```

4.5.5 CapsuleRoleCompliance

The CapsuleRoleCompliance relation defines compliance between a *design* CapsuleRole and a *tSpec* CapsuleRole. A *design* CapsuleRole is considered compliant with a *tSpec* CapsuleRole if their name and the Capsule they are a projection of are the same.

Dependent on	∅
Rules dependent on it	4.5.6 CapsuleStructureCompliance

4.5.5.1 QVT Relation Definition

```

relation CapsuleRoleCompliance
{
  domain { (CapsuleRole) [name = n, capsule = cSpec] }
  domain { (CapsuleRole) [name = n, capsule = cDesign] }
  when
  {
    cSpec.name = cDesign.name
  }
}

```

4.5.6 CapsuleStructureCompliance

The CapsuleStructureCompliance relation defines compliance between a RoseRT *design* CapsuleStructure element and a RoseRT *tSpec* CapsuleStructure element. The *design* element is considered compliant if its classifierRoles set contains a compliant ClassifierRole for each of the ClassifierRoles in the *tSpec* element, the same with the connectors and ports sets.

Dependent on	4.5.5 CapsuleRoleCompliance 4.5.8 ConnectorCompliance 4.5.13 PortCompliance
Rules dependent on it	4.5.4 CapsuleCompliance

4.5.6.1 QVT Relation Definition

```

relation CapsuleStructureCompliance
{
  domain
  {
    (CapsuleStructure) [ classifierRoles = crSpec, connectors = conSpec, ports = pSpec ]
  }
  domain
  {
    (CapsuleStructure) [ classifierRoles = crDesign, connectors = conDesign, ports = pDesign ]
  }
  when
  {
    crSpec->forAll(cr | CapsuleRoleCompliance(cr, crDesign)) and
    conSpec->forAll(con | ConnectorCompliance(con, conDesign)) and
    pSpec->forAll(p | PortCompliance(p, pDesign))
  }
}

```


4.5.7 ClassCompliance

The ClassCompliance relation defines the compliance rules between a RoseRT *design* Class element and a RoseRT *tSpec* Class element. A *design* Class is considered compliant with the *tSpec* element if its name is the same as the *tSpec* name, its attribute set contains at least one compliant Attribute for each of the attributes in the *tSpec* attribute set, the same between the two operation sets and generalization sets. That is, the *design* model may define additional attributes, operations and generalizations. Attribute compliance is verified by the AttributeCompliance rule, Operation compliance by the OperationCompliance rule, Generalization compliance by the GeneralizationCompliance.

Dependent on	4.5.3 AttributeCompliance 4.5.10 OperationCompliance 4.5.9 GeneralizationCompliance
Rules dependent on it	4.5.11 PackageCompliance

4.5.7.1 QVT Relation Definition

```

relation ClassCompliance
{
  domain {(Class) [ name = n, attributes = aSpec, operations = oSpec, generalizations = gSpec ]}
  domain {(Class) [ name = n, attributes = aDesign, operations = oDesign, generalizations = gDesign]}
  when
  {
    aSpec->forAll(a | AttributeCompliance(a, aDesign)) and
    oSpec->forAll(o | OperationCompliance(o, oDesign)) and
    gSpec->forAll(g | GeneralizationCompliance(g, gDesign))
  }
}

```

4.5.8 ConnectorCompliance

The ConnectorCompliance relation defines compliance between a *design* Connector element and a *tSpec* Connector element. The *design* element is considered compliant if it shares the same name with the *tSpec* element and the two Ports that the Connector connects are compliant.

Dependent on	4.5.13 PortCompliance
Rules dependent on it	4.5.5 CapsuleRoleCompliance

4.5.8.1 QVT Relation Definition

```

relation ConnectorCompliance
{
  domain { (Connector) [name = n, port1 = p1Spec, port2 = p2Spec] }
  domain { (Connector) [name = n, port1 = p1Design, port2 = p2Design] }
  when
  {
    PortCompliance(p1Spec, p1Design) and PortCompliance(p2Spec, p2Design)
  }
}

```

4.5.9 GeneralizationCompliance

The GeneralizationCompliance relation defines compliance between a *design* Generalization element and a *tSpec* Generalization element. A *design* element is considered compliant if it shares the same supplierClassifier as the *tSpec* element.

Dependent on	$\emptyset$
Rules dependent on it	4.5.4 CapsuleCompliance 4.5.7 ClassCompliance 4.5.14 ProtocolCompliance

4.5.9.1 QVT Relation Definition

```

relation GeneralizationCompliance
{
  domain {(Generalization)[ supplierClassifier = scSpec]}
  domain {(Generalization)[ supplierClassifier = scDesign]}
  when
  {
    scSpec.name = scDesign.name
  }
}

```

4.5.10 OperationCompliance

The OperationCompliance relation defines compliance between a *design* Operation element and a *tSpec* Operation element. A *design* element is considered compliant with a *tSpec* element if:

- the design element shares the same name as the tSpec element;
- the design element shares the same returnType as the tSpec element; and

- the design parameters set contains a compliant Parameter element for each Parameter element contained in the tSpec parameters set.

Dependent on	4.5.12 ParameterCompliance
Rules dependent on it	4.5.4 CapsuleCompliance 4.5.7 ClassCompliance

4.5.10.1 QVT Relation Definition

```

relation OperationCompliance
{
  domain {(Operation)[name = n, returnType = rt, parameters = pSpec]}
  domain {(Operation)[name = n, returnType = rt, parameters = pDesign]}
  when
  {
    pSpec->size() = pDesign->size()
    and
    pSpec->forall(p | ParameterCompliance(p, pDesign))
  }
}

```

4.5.11 PackageCompliance

The PackageCompliance relation defines compliance between a RoseRT *design* LogicalPackage element and a RoseRT *tSpec* element. The *design* element is considered compliant if:

- the design element shares the same name as the tSpec element;
- the design logicalPackages set contains at least one compliant LogicalPackage element for each LogicalPackage element contained in the tSpec logicalPackages set;
- the design classes set contains at least one compliant Class element for each Class element contained in the tSpec classes set;
- the design capsules set contains at least one compliant Capsule element for each Capsule element contained in the tSpec capsules set;
- the design protocols set contains at least one compliant Protocol element for each Protocol element contained in the tSpec protocols set; and

- the design associations set contains at least one compliant Association element for each Association element contained in the tSpec associations set.

Dependent on	4.5.7 ClassCompliance 4.5.4 CapsuleCompliance 4.5.14 ProtocolCompliance 4.5.1 AssociationCompliance
Rules dependent on it	4.5.11 PackageCompliance

4.5.11.1 QVT Relation Definition

relation PackageCompliance

```

{
  domain
  {
    (LogicalPackage)
    [
      name = n,
      logicalPackages = lpSpec,
      classes = cSpec,
      capsules = capSpec,
      protocols = pSpec,
      associations = aSpec
    ]
  }
  domain
  {
    (LogicalPackage)
    [
      name = n,
      logicalPackages = lpDesign,
      classes = cDesign,
      capsules = capDesign,
      protocols = pDesign,
      associations = aDesign
    ]
  }
  when
  {
    lpSpec->forAll(lp | PackageCompliance(lp, lpDesign)) and
    cSpec->forAll(c | ClassCompliance(c, cDesign)) and
    capSpec->forAll(cap | CapsuleCompliance(cap, capDesign)) and
    pSpec->forAll(p | ProtocolCompliance(p, pDesign)) and
    aSpec->forAll(a | AssociationCompliance(a, aDesign))
  }
}

```

4.5.12 ParameterCompliance

The ParameterCompliance relation defines compliance between a *design* Parameter element and a *tSpec* Parameter element. A *design* element is considered compliant if it shares the same name and type with the *tSpec* element.

Dependent on	$\emptyset$
Rules dependent on it	4.5.10 OperationCompliance

4.5.12.1 QVT Relation Definition

```

relation ParameterCompliance
{
  domain {(Parameter)[ name = n, type = t]}
  domain {(Parameter)[ name = n, type = t]}
}

```

4.5.13 PortCompliance

The PortCompliance relation defines compliance between a *design* Port element and a *tSpec* Port element. A *design* element is considered compliant with a *tSpec* element if they share the same value for the name, conjugated and protocol features. The protocol feature is considered the same if the name of the protocol is the same.

Dependent on	$\emptyset$
Rules dependent on it	4.5.5 CapsuleRoleCompliance 4.5.8 ConnectorCompliance

4.5.13.1 QVT Relation Definition

```

relation PortCompliance
{
  domain {(Port)[name = n, conjugated = conj, protocol = pSpec]}
  domain {(Port)[name = n, conjugated = conj, protocol = pDesign]}
  when
  {
    pSpec.name = pDesign.name
  }
}

```

4.5.14 ProtocolCompliance

The ProtocolCompliance relation defines compliance between a *design* Protocol element and a *tSpec* Protocol element. A *design* element is considered compliant with a *tSpec* element if:

- the design name is the same as the tSpec name;
- the design generalizations set contains at least one compliant Generalization element for each Generalization element in the tSpec generalizations set;
- the design inSignals set contains at least one compliant Signal element for each Signal element in the tSpec inSignals set; and
- the design outSignals set contains at least one compliant Signal element for each Signal element in the tSpec outSignals set.

Dependent on	4.5.9 GeneralizationCompliance 4.5.15 SignalCompliance
Rules dependent on it	4.5.11 PackageCompliance

4.5.14.1 QVT Relation Definition

relation ProtocolCompliance

```
{
  domain
  {
    (Protocol) [ name = n, generalizations = gSpec, inSignals = inSpec, outSignals = outSpec ]
  }
  domain
  {
    (Protocol)[ name = n, generalizations = gDesign, inSignals = inDesign, outSignals = outDesign ]
  }
  when
  {
    gSpec->forAll(g | GeneralizationCompliance(g, gDesign)) and
    inSpec->forAll(i | SignalCompliance(i, inDesign)) and
    outSpec->forAll(o | SignalCompliance(o, outDesign))
  }
}
```

4.5.15 SignalCompliance

The SignalCompliance relation defines compliance between a *design* Signal element and a *tSpec* Signal element. A *design* element is considered compliant with a *tSpec* element if it shares the same name and type.

Dependent on	$\emptyset$
Rules dependent on it	4.5.14 ProtocolCompliance

4.5.15.1 QVT Relation Definition

```

relation SignalCompliance
{
  domain {(Signal)[ name = n, type = t]}
  domain {(Signal)[ name = n, type = t]}
}

```

4.6 Compliance Rule Verification

In this section we provide the test models for two of the compliance rules defined in the previous section. For each of the compliance rules we provide the combinatorial we have developed to verify the rule. A project must be created specifically to verify each rule: this is a limitation of the prototype. For testing a particular rule we first develop the combinatorial model from which a test model is built. The *specification* model for the project we have created is set to the test model. The test model is a set of test cases where each test case has an identifier a *tSpec* element and a *design* element. In order to evaluate the compliance rule under test (CRUT) we set the mapping starting point to be a test driver. Within the mappings section of the project the CRUT and supporting rules are included. The supporting rules can be determined from the compliance rule dependency graph that is provided in Figure 78. The two compliance rules that the combinatorial model is provided for are *ClassCompliance* and *AttributeCompliance*. We chose one rule that contains sets of elements (ClassCompliance) and a second rule, which did not contain any sets of elements (AttributeCompliance).

4.6.1 Verifying the AttributeCompliance rule

For the AttributeCompliance relation to produce a success result the name and type of both input Attributes must be the same. Therefore, to verify that this is in fact the result that is produced we execute the following test cases against the relation.

Table 17 Combinatorial model that defines the test cases for verifying the AttributeCompliance rule

Test Case	tSpec input		design input		Expected Result
	name	type	name	type	
1	η	τ	$= \eta$	$= \tau$	success
2	η	τ	$\neq \eta$	$\neq \tau$	fail
3	η	τ	$\neq \eta$	$= \tau$	fail
4	η	τ	$\neq \eta$	$\neq \tau$	fail
5	η	"	$= \eta$	$= "$	success
6	η	"	$= \eta$	$\neq "$	fail
7	η	"	$\neq \eta$	$= "$	fail
8	η	"	$\neq \eta$	$\neq "$	fail
9	"	τ	$= "$	$= \tau$	success
10	"	τ	$= "$	$\neq \tau$	fail
11	"	τ	$\neq "$	$= \tau$	fail
12	"	τ	$\neq "$	$\neq \tau$	fail
13	"	"	$= "$	$= "$	success
14	"	"	$= "$	$\neq "$	fail
15	"	"	$\neq "$	$= "$	fail
16	"	"	$\neq "$	$\neq "$	fail

Table 18 Explanation of symbols used in Table 17.

Symbol	Explanation
η	A value for name, that is not the empty-string and held constant for a particular row.
τ	A value for type, that is not the empty-string and held constant for a particular row.
"	Empty-string
$\neq$	Not equal to the value to its right. For example, $\neq \tau$ means not equal to the value for type in the tSpec input object.
$=$	equal to the value to its right. For example, $= \tau$ means equal to the value for type in the tSpec input object.

Ideally, when the compliance rule fails it would indicate the exact cause of the failure. However, when we execute the test driver a set of test case results are produced. Each test case result

identifies the test case and the result of evaluating the CRUT for that particular test case. That is, if executing test case 2, it will not specify that *name* feature caused the failure, only that the results of evaluating the test case results in a failure.

```

mapping AttributeComplianceTestDriver(testCases)
{
  domain { (AttributeTest)[ tests = attTests ]}
  body
  {
    attTests->collect(t | AttributeCompliance(t.Spec, t.Design))
  }
}

```

Figure 79 AttributeCompliance rule test driver

In order to verify that the compliance rule evaluated as intended, two instantiations of the combinatorial model were developed. In the instantiation, for each row in the combinatorial model (Table 17) a *tSpec* and *design* object along with the test case identifier were created with the values specified by the row. When the test driver shown in Figure 79 was executed with each of these test models the expected results were produced.

4.6.2 Verifying the ClassCompliance rule

In the previous section the verification of a compliance rule was somewhat simplistic. However, it illustrated the approach. The ClassCompliance rule involves sets, namely the attributes, operations and generalizations features of a Class. In building the combinatorial model two aspects of the set must be considered; its size and the contents of the set. For the purpose of the set contents we are concerned with how the elements of the *design* set relate to the elements of the corresponding *tSpec* set. Tests must evaluate *tSpec* and *design* sets that are equal, a *tSpec* set that is a proper subset of the *design* set and a *design* set that is a proper subset of the *tSpec* set. Two elements of a set are considered equal if they are compliant. Tests must evaluate each of these cases with sets of different sizes: empty, one element (singleton) and more than one element. Relying on the insights of Chapter 6 in [2], combined with our analysis in this paragraph results in the

combinations shown in Table 19. This model shows that we have eleven (11) valid tests for a single set²⁶.

Table 19 Set testing combinations. Rows marked illegal are due to the definition of the proper subset operator

Set elements	tSpec size	design size	
tSpec = design	empty	empty	
	1	1	
	> 1	> 1	
tSpec $\subset$ design	empty	empty	illegal
	empty	1	
	empty	> 1	
	1	empty	illegal
	1	1	illegal
	1	> 1	
	> 1	> 1	
tSpec $\supset$ design	empty	empty	illegal
	1	empty	
	> 1	empty	
	empty	1	illegal
	1	1	illegal
	> 1	1	
	> 1	> 1	

The ClassCompliance rule has three sets and the name feature: if we tested all possible combinations this would result in at least 2662 ($2 * 11^3$) test cases. Instead, taking inspiration from Chapter 6 of [2], we suggest building a test model to test a particular feature at time. That is, if name is under test then the three sets would always be compliant and if attributes were under test then name and the other two sets would be compliant, and so on. Making use of Table 19 we arrive at the following combinatorial model to test the ClassCompliance relation.

²⁶ We do not claim that model is complete only that it is the result of our initial analysis. Further analysis is required to ensure that all cases are covered.

Table 20 Combinatorial model for testing the ClassCompliance relation. The symbol 't' represents tSpec and 'd' design. A checkmark signifies a success and an X a failure

	name	attribute			operations			generalizations			result
		set elements	tSpec size	design size	set elements	tSpec size	design size	set elements	tSpec size	design size	
1	equal	compliant			compliant			compliant			✓
2	not equal	compliant			compliant			compliant			X
3	equal	t = d	0	0	compliant			compliant			✓
4	equal	t = d	1	1	compliant			compliant			✓
5	equal	t = d	> 1	> 1	compliant			compliant			✓
6	equal	t ⊂ d	0	1	compliant			compliant			✓
7	equal	t ⊂ d	0	> 1	compliant			compliant			✓
8	equal	t ⊂ d	1	> 1	compliant			compliant			✓
9	equal	t ⊂ d	> 1	> 1	compliant			compliant			✓
10	equal	t ⊃ d	1	0	compliant			compliant			X
11	equal	t ⊃ d	> 1	0	compliant			compliant			X
12	equal	t ⊃ d	> 1	1	compliant			compliant			X
13	equal	t ⊃ d	> 1	> 1	compliant			compliant			X
14	equal	compliant			t = d	0	0	compliant			✓
15	equal	compliant			t = d	1	1	compliant			✓
16	equal	compliant			t = d	> 1	> 1	compliant			✓
17	equal	compliant			t ⊂ d	0	1	compliant			✓
18	equal	compliant			t ⊂ d	0	> 1	compliant			✓
19	equal	compliant			t ⊂ d	1	> 1	compliant			✓
20	equal	compliant			t ⊂ d	> 1	> 1	compliant			✓
21	equal	compliant			t ⊃ d	1	0	compliant			X
22	equal	compliant			t ⊃ d	1	0	compliant			X
23	equal	compliant			t ⊃ d	> 1	1	compliant			X
24	equal	compliant			t ⊃ d	> 1	> 1	compliant			X
25	equal	compliant			compliant			t = d	0	0	✓
26	equal	compliant			compliant			t = d	1	1	✓
27	equal	compliant			compliant			t = d	> 1	> 1	✓
28	equal	compliant			compliant			t ⊂ d	0	1	✓
29	equal	compliant			compliant			t ⊂ d	0	> 1	✓
30	equal	compliant			compliant			t ⊂ d	> 1	> 1	✓

Table 21 Combinatorial model from Table 20 continued

	name	attribute			operations			generalizations			result
		set elements	tSpec size	design size	set elements	tSpec size	design size	set elements	tSpec size	design size	
31	equal	compliant			compliant			$t \subset d$	> 1	> 1	✓
32	equal	compliant			compliant			$t = d$	1	1	X
33	equal	compliant			compliant			$t \supset d$	> 1	0	X
34	equal	compliant			compliant			$t \supset d$	1	1	X
35	equal	compliant			compliant			$t \supset d$	> 1	> 1	X
36	not equal	not compliant			not compliant			not compliant			X

In order to verify that the compliance rule evaluated as intended, one instantiation of the combinatorial model was developed. The instantiation contains one tSpec input object and design input object pair for each row in the model. The object's values reflect the values defined in its respective row of the model. When the test driver shown in Figure 80 was executed with each of these test models the expected results were produced.

```

mapping Class ComplianceTestDriver(testCases)
{
  domain { (ClassTest)[ tests = clsTest ]}
  body
  {
    (Class ComplianceTestResults)
    [
      results = clsTest->collect(t |
        (TestCaseResult)
        [
          testcase = t.testcase,
          result = Class Compliance(t.Spec, t.Design)
        ]
      )
    ]
  }
}

```

Figure 80 A test driver for testing the ClassCompliance relation against a set of test

4.7 Evaluation of Compliance

In this section we apply the mappings and rules to a *specification* model and *design* model. The *specification* model is of our own invention. It is a fictional *specification* model for a Bank system server influenced by the ATM system in [21]. The *specification* is modeled in IBM Rational Rose and consists of a root Package BankSystem that contains four Packages: ActiveClasses, BaseTypes, DataClasses, Interfaces and SequenceDiagrams. Similarly a BankSystem *design* was construct in IBM Rational Rose Real-Time, which follows the same package structure. The *design* is more concrete than the *specification* and thus may contain additional elements.

We conclude the evaluation of compliance with a look at the execution of the mappings to generate the *tSpec* model and the evaluation of the compliance rules. More specifically, we examine the *tSpec* model produced by applying the mappings; this includes the traceability model as well as the mapping statistics. The analysis of the compliance rule evaluation includes a discussion of the evaluation against a *design* that is compliant and one that is non-compliant.

4.7.1 Specification Model

The *specification* model is a fictitious *specification* for Bank server system. The *specification* is a UML 1.5 model consisting of classes, interfaces, attributes, operations, class relationships and a set of sequence diagrams. In Table 22, a list of the element types that appear in the *specification* is provided along with the number of times they occur. For example, the *specification* model contains 35 elements of type Class.

Table 22 *Specification* model element types and the number of occurrences in the model

Element Type	Count
Association	35
AssociationEnd	70
Attribute	27
Class	35
Generalization	15
Interaction	15
Interface	2
Message	78
Operation	59
Package	6
Parameter	142

The package, class and sequence diagrams for the *specification* model are shown in Appendix E. The Class elements in the ActiveClasses package are those Class elements that will be mapped into Capsules in the *tSpec* model. Therefore, the name of each class must be added to the *active* set in the additional information model. All of the passive Class elements are contained in the DataClasses package, they will be mapped into Class elements in the *tSpec* model. Similarly, the BaseType package contains a set of primitive types and collection types that are used within the *specification*, they are also mapped into Class elements in the *tSpec* model. The sequence diagrams package contains a set of 15 sequence diagrams that describe the interaction patterns between the active classes in the model. They are used by the mappings in order to generate the signals for the Protocol elements. Finally, the Interfaces package contains results of mapping the Interface elements defined in the *specification* model, which are realized by two classes that will be mapped into Capsules.

4.7.2 Design Model

The case study's *design* is a realization of the Bank server *specification*. The difference is that the *design* uses the RoseRT toolset modeling language. The *design* model is shown in detail in Appendix F. The compliance rules we use require that the *design* adhere to the package structure of the

specification model. As a consequence the logical structure of the *design* model is not ideal. A possible solution would be to relax the package structure restriction in the *design*.

The *design* specifies an additional Capsule element that is the actual Bank server entity; it contains CapsuleRole elements for each of the Capsules that are expected to be compliant with the *specification* model. Additionally the *design* model does not specify any sequence diagrams. The sequence diagrams in the *specification* model show the communication between active classes. In the context of this case study, we use the *specification* sequence diagrams for extracting structural information, namely the signals required for a particular Protocol. Therefore, we do not map the sequence diagrams to the *tSpec* model or expect them to appear in the *design* model.

The *design* model contains 6 logical packages, 51 classes, 12 capsules and 27 protocols. The additional classes are a product of mapping Message elements from UML to Signal elements in RoseRT. A Signal can have a single parameter, whereas, the UML Operation element that corresponds with a Message can have multiple parameters. As required by the compliance rules a Class element must be created for a Signal mapped from a Message that carries more than one piece of data within. These classes are found in the Sequence Diagrams package. Class elements were also created for Interface operations found in the *specification* for the same reason.

4.7.3 Generated tSpec model

In this section we look at the products of applying the mappings defined in Section 4.3 to the *specification* model. More specifically, this section examines the mapping statistics produced by the execution, the *tSpec* model created, the traceability model and the completeness of the mappings.

The UMLToRoseRT mapping was the starting point, which produces a RoseRT *tSpec* model. Either the UMLToRoseRT mapping or one of the other support mappings invokes each of the supporting mappings. The data in Table 23 gives the mapping statistics produced by the prototype. When these statistics are compared to the model statistics found in Table 22, some mappings are executed more than expected. For example PackageToLogicalPackage is invoked 7 times when there are 6 Package elements in the *specification* model. This results from the input

model having an additional top-level package in which all model elements are contained. That is, an additional Package element is added to the model to represent the model itself. Similarly, `OperationResultToSignal` is invoked 86 times when there are only 59 Operation elements in the *specification*. This results from more than one message in a sequence diagram referring to the same element. Duplicate signals do not however appear in the *tSpec* model as the mapping definitions prevent duplicate signals being added to a Protocol element.

A *specification* Class element is either mapped into a Capsule element or a Class element and therefore we expect that the sum of the `ClassToDataClass` and `ClassToCapsule` mappings be equal to the number of Class elements in the *specification*. Analyzing our results `ClassToDataClass` is invoked 24 times and `ClassToCapsule` 11 times which sums to 35 the number of Class elements in the *specification* model. A software engineer can perform this type of analysis on the *specification* model and the mapping statistics in order to gain confidence in the *tSpec* model.

Table 23 Case study mapping application statistics

Mapping	Success	Failure
UMLToRoseRT	1	0
PackageToLogicalPackage	7	0
ClassToDataClass	24	0
AttributeToAttribute	27	0
OperationToOperation	37	0
ParameterToParameter	64	0
GeneralizationToGeneralization	13	0
InterfaceToDataClasses	2	0
OperationToDataClass	78	0
ParameterToAttribute	138	0
AssociationToProtocol	26	0
AssociationEndsToProtocolName	26	0
InterfaceToProtocol	2	0
InterfaceToOperationSignal	12	0
OperationResultToSignal	86	0
InteractionToDataClasses	15	0
ClassToCapsule	11	0
AssociationToPort	44	0
AssociationEndsToPortName	44	0
GeneralizationToPort	2	0
InterfaceToPortName	2	0
AssociationToAssociation	9	0
AssociationEndToAssociationEnd	18	0
CollectProtocols	7	0
ProcessInteractions	7	0
InteractionAddToProtocol	15	0
MessageToSignal	74	0
OperationToSignal	74	0

Another product of executing the mappings is the traceability model. Recall the traceability model is produced by the prototype upon completion of the *specification* model to *tSpec* model transformation. The traceability model produced by applying the UMLToRoseRT mapping to the *specification* model contains 891 trace nodes and is 8320 lines. Due to the length of the traceability

model we do not include it in the dissertation. Instead in Appendix G we include an example for each of the mappings. While the traceability model is large accessing the desired information can be achieved using XPath [55]. For example, in any of the XML IDE's (for example Stylus Studio 6 [11]) available the statement `./trace/mapping-applied [. = "UMLToRoseRT"]` will retrieve all of the trace elements for the UMLToRoseRT mapping. Furthermore, the statement `./trace[target-objects/obj/@type = "Capsule"]` will retrieve all traceability items where the target element was a Capsule.

The traceability model discussed in the previous paragraph is used by the prototype to produce another artefact; the completeness analysis. The completeness analysis is a list of all of the elements (type of element and unique identifier) in the model as well as the number of times each one was a source element in a mapping. As previously discussed in Section 3.1.2.1 this artefact is derived by analyzing both the *specification* model to retrieve all of the elements and processing the traceability model to count the number times an element was the source of a mapping. There are 484 elements in the model and therefore the whole artefact cannot be included. However, analyzing the artefact reveals the elements in Table 24 do not appear in the traceability model.

Table 24 Completeness analysis artefact

Specification Element	# times used in a mapping
Message (4197F9140324)	0
Message (4197F945039F)	0
Message (4197F96D02CB)	0
Message (4198A6EB013F)	0
Parameter (022cf9a7-de3b-4ee2-b55c-e03227960305)	0
Parameter (02e1f06c-e08a-4766-aaa9-70acd7fc0582)	0
Parameter (06423900-a35b-4449-9060-05cd80cd9942)	0
Parameter (090da1d5-e18f-4154-bccc-b32819f468c2)	0
Parameter (09f7b7be-c62e-463d-ad19-30635286ba15)	0
Parameter (0ad02199-49bf-446e-935c-f19280b96a52)	0
Parameter (0ccde004-6058-47e2-8ce8-2f8e2091fb44)	0
Parameter (0d74bb3c-ebc2-4859-b7f6-bcd6b0314d53)	0
Parameter (0e31a36a-4e58-4469-80fe-28a2576e74a8)	0
Parameter (11b46391-bef0-41aa-9604-3e5bfb623817)	0

Parameter (11bcda56-bf2b-4cae-976a-d7af80bfef18)	0
Parameter (1a649b31-ce87-47fd-a549-e673d1d6cbee)	0
Parameter (20a4deaa-ba8f-4ac6-a86b-f7b03409d47c)	0
Parameter (232033e2-45f4-4620-94e4-ec82f1a9a83a)	0
Parameter (25314e36-8c82-468b-92a3-2d7ebf760af7)	0
Parameter (2fb85b08-58c9-485a-a898-0e0363d5eaa8)	0
Parameter (31ab68bd-4ea3-401a-b8aa-5df7e7f11523)	0
Parameter (345ad84b-fe71-4d19-a900-3f9471d55260)	0
Parameter (34862a76-bc80-4d59-a543-fa238dfe11a9)	0
Parameter (372f5152-06a1-4390-8c12-84c521efafb9)	0
Parameter (3ba72a6c-113c-48e4-aaba-67c3f588f595)	0
Parameter (4cf52add-7eaf-4bf1-91f7-15ecbf70f6f5)	0
Parameter (4e30e65c-2059-4ce8-b251-45c984a80669)	0
Parameter (4f3877fa-fcb1-4d3b-b784-0c664b5bd48a)	0
Parameter (514cd59c-94cf-4235-b9e5-f95f24a74024)	0
Parameter (571f1a87-5708-4dc5-8f0d-f7df6d184a2b)	0
Parameter (57ea1886-551f-4a0e-b6f1-cc2c16b5dd68)	0
Parameter (5c7fe469-d5dc-4262-aaf3-1798b266e24a)	0
Parameter (614a0227-6880-40d0-b45e-93601fad6aee)	0
Parameter (63210145-dbc3-44c7-8591-3c6a6a2f0a55)	0
Parameter (694ab5f5-c943-45df-b59f-0cce78087daf)	0
Parameter (699983cc-07c3-47a4-9c9a-95469a1bbde1)	0
Parameter (6ac68824-c3a6-4a6e-ac06-ca4af0d61ff1)	0
Parameter (7c6e92af-bbb5-431e-9f1b-c9cd7736c77d)	0
Parameter (7fa60c82-cac4-449b-8752-7dcd6fe04f9c)	0
Parameter (82e79e56-85e8-4a8b-9f7b-94d94c6d2255)	0
Parameter (868348da-2ced-49f0-83fc-43604e92a68a)	0
Parameter (889831e9-cd52-4d3e-b34c-f3d64c66b130)	0
Parameter (932f0248-3186-4753-bcb8-5bdfac0dcfc1)	0
Parameter (96f9678c-86d1-4af3-bb61-c1d49a4c782a)	0
Parameter (9c3e1473-61a2-44b4-b5aa-c41769c1b084)	0
Parameter (a5d913e3-c48e-443d-b205-b06513da7f1d)	0
Parameter (b09ebda4-2652-4f22-919d-e7d5ea3a631e)	0
Parameter (b76ebe39-7005-492a-90e9-eada8fd51640)	0
Parameter (baf45d95-9884-42ee-93b2-404ad274f266)	0
Parameter (bb48c4c1-f0f5-498b-b282-bb299bf8ca20)	0
Parameter (bc416d50-a9d7-4e40-8713-eb1220050c68)	0
Parameter (c4904f85-558b-411c-9f63-46021081d3ec)	0
Parameter (c9609adf-6e8c-4c65-ab99-924f57db2e0c)	0
Parameter (cd5c10a5-3e94-49d5-9206-5012b34ab857)	0

Parameter (db045370-4c4c-41c9-84a8-f31eb4df63f9)	0
Parameter (dba54be9-f926-410d-a60f-52faec3723f4)	0
Parameter (e3cb5421-cad7-4a8b-9f40-0753ffc37a00)	0
Parameter (ed033e27-2319-4d26-956e-61c7e5afc432)	0
Parameter (cef34981-5b94-4481-8bec-fd0c637aa9ed)	0
Parameter (efa81843-11e6-4448-951b-602430f6b214)	0
Parameter (f1da6fce-5049-4b57-8cf6-f0b236556982)	0
Parameter (f58f17de-3a4c-4271-89c0-5fcd21b6d74a)	0
Parameter (feedc9e3-257e-4ad7-af02-161c7dc2da53)	0

Examining the Message elements that were not mapped reveals that they are Messages in the *specification* model that were not linked to Operations of the receiving Class element. The *specification* model was corrected and the transformations re-executed. In the re-execution these Message elements were used as the source of at least one mapping.

The Parameter elements found in Table 24 are all return parameters. That is, the kind field of the Parameter is 'return'. Furthermore, when the number of Parameter elements in the table is counted, exactly 59 Parameters of type 'return' did not appear in the traceability model. This happens to be the number of operations in the *specification* model. The relationship between the number of Parameters and the number of Operations alludes to an issue with mapping the return Parameter of an Operation. Examining the mapping it is evident that the return Parameter is selected within the OperationToOperation mapping. As we mentioned in Section 3.1, a current limitation of the QVT engine is that traceability items are reported at the mapping level and since the Parameter was not the input to mapping it is not reported in the traceability model. This is, we repeat, an issue that will be addressed by future work. In order to correct this, a new mapping can be created that maps the return Parameter element to the return type for the RoseRT model (*tSpec*). In an effort to ensure that this was the issue, the new mapping was created and the *specification* to *tSpec* transformation was re-executed. The re-execution resulted in all elements of the *specification* model being used at least once by a mapping.

The main product of the *specification* model to *tSpec* model transformation is the *tSpec* model. In order to ensure that the model that is produced can be used to create a valid RoseRT model, the RoseRT Model Converter write operation is invoked with the generation *tSpec* model as the input.

The *tSpec* that was produced was successfully passed to the Model Converter and the resulting RoseRT model could be opened without error in RoseRT. In other words, the *tSpec* model for the case study was successfully created.

4.7.4 Verifying Design Compliance

The previous section discussed the creation of the *tSpec* model from the *specification* model. Since the *tSpec* model has been created and the compliance rules defined, a *design* model can be verified for compliance. Section 4.7.2 presented the *design* model, which will be verified for compliance. Evaluating the compliance rules against the *design* model reveals that it is in fact compliant. This however is not interesting; it is more interesting to know that if a *design* is not compliant that the prototype will in detect such a situation. We evaluate in this section a *design* model this is missing a Package element and a second *design* model missing an Attribute in a Class.

Suppose the *design* model is missing the Package element BaseTypes. When the compliance rules are evaluated the prototype indicates that the *design* model is not compliant with the *tSpec* model. Additionally, the compliance failure traceability model shown in Figure 81 is produced to help the software engineer in fixing their *design* model. The model shows that the BaseTypes package in the BankSystem package in the Model cannot be found in the *design*.

```

<?xml version="1.0" encoding="utf-16"?>
<trace-info>
  <trace-item mapping="PackageConformance" e-name="Model" e-type="LogicalPackage"
    e-id="9075e136a94b" result="FAIL">
    <children>
      <trace-item mapping="PackageConformance" e-name="BankSystem" e-type="LogicalPackage"
        e-id="2817108157c2" result="FAIL">
        <children>
          <trace-item mapping="PackageConformance" e-name="BaseTypes"
            e-type="LogicalPackage" e-id="b3caf7ba60e9" result="FAIL">
            <children />
          </trace-item>
        </children>
      </trace-item>
    </children>
  </trace-item>
</trace-info>

```

Figure 81 Compliance failure traceability model produced when design is missing the BaseTypes package

Now suppose that Person class in the *design* model is missing the creditRating attribute. When the compliance rules are evaluated the model shown in Figure 82 is produced. This information allows the software engineer to locate the element in the *iSpec* model in order to gather the information necessary to fix the *design* model in order to make it compliant.

These two examples show of information that is produced when a *design* is missing an element that the *specification* model has defined. We leave to future work the task of increasing the quality of the information produced when compliance fails. For example, currently the prototype does not provide a specific reason for an elements failure. Perhaps the creditRating attribute exists in the *design* but with the incorrect type. In order to increase the effectiveness of the prototype, such information could be provided to make the software engineers life easier when debugging the *design* model.

```

<?xml version="1.0" encoding="utf-16"?>
<trace-info>
  <trace-item mapping="PackageCompliance" e-name="Model" e-type="LogicalPackage"
    e-id="9075e136a94b" result="FAIL">
    <children>
      <trace-item mapping="PackageCompliance" e-name="BankSystem" e-type="LogicalPackage"
        e-id="2817108157c2" result="FAIL">
        <children>
          <trace-item mapping="PackageCompliance" e-name="DataClasses"
            e-type="LogicalPackage" e-id="b3caf7ba60e9" result="FAIL">
            <children>
              <trace-item mapping="ClassCompliance" e-name="Customer"
                e-type="Class" e-id="419020B002A3" result="FAIL">
                <children>
                  <trace-item mapping="AttributeCompliance" e-name="creditRating"
                    e-type="Attribute" e-id="d02ea9946239" result="FAIL">
                    </children>
                  </trace-item>
                </children>
              </trace-item>
            </children>
          </trace-item>
        </children>
      </trace-item>
    </children>
  </trace-item>
</trace-info>

```

Figure 82 Non-compliance information produced when design is missing the creditRating attribute in the Customer class

Chapter 5

CONCLUSION AND FUTURE WORK

This chapter will outline various directions for future work. Following the outline for future work, the contributions made by this work are discussed. The chapter will finish with some concluding statements.

5.1 Contributions

- A. In Chapter 3 we show that a model-to-model transformation language, QVT, can be used to support the activity of model compliance verification. Furthermore, we evaluate a design for compliance with a specification in Chapter 4, using QVT as the mapping and compliance rule definition language.
- B. In Section 3.2, we discuss how the prototype we propose in this dissertation provides a more general solution to the activity of model compliance verification than the tools defined in [70].
- C. The case study in Chapter 4 defines a set of mappings for transforming a UML 1.5 model into a RoseRT model. Either the overall approach or a subset of the mappings defined could be re-used or extended for transforming a different UML 1.5 *specification* into a RoseRT *tSpec*.
- D. The case study in Chapter 4 defines a set of compliance rules for verifying compliance between a RoseRT *tSpec* and a RoseRT design. These compliance rules can be re-used or extended for verifying compliance in a different project or case study.

- E. In Chapter 3 and Chapter 4 we show that QVT mappings and relations can be used to unit test a QVT mapping or relation.

5.2 Future Work

Future work for this dissertation falls into two categories. The first category is future work related to the proposed prototype. The second category discusses future work related to the QVT.

5.2.1 Prototype Future Work

In its current state our proposed prototype requires a significant level of effort to verify the mappings and compliance rules. Increasing the level of automation provided for these verification activities requires further research. Our goal is to allow a software engineer to define their test suites in the form of a combinatorial model. The prototype would automate the verification of the mappings and compliance rules from the combinatorial model.

The proposed prototype currently reports traceability at the level of mapping parameters. That is, the source elements of a target element are reported as the elements passed into the mapping. There is room for improvement as the mapping may select objects contained within one of the inputs and make direct use of this object in creating the target object. Therefore, we need to investigate the level of granularity necessary in the traceability model, in order to make the software engineers life easier.

We have not explored the support for compliance between a model and its modeling language. That is, we can consider the metamodel for a modeling language to be a specification for a model. We feel that being able to verify that the model complies with the semantics defined in its metamodel is important and is part of our future considerations. This research will include, in particular, verifying that a stereotyped element complies with the semantics of the stereotype.

Finally, the case study presented in this dissertation was good enough to show the feasibility of supporting model compliance verification. It does not however, show the scalability of our proposed prototype for model compliance verification. Therefore, we must investigate the use of larger specifications and hopefully specification and design models from industry. Another evaluation of scalability is to use a standard that is defined by a standardization body like the OMG or ISO as the *specification* model.

5.2.2 QVT Related Future Work

As we described in Section 3.1.1 the QVT engine (interpreter) we have built into our prototype is not optimized. By optimizing the QVT interpreter we will reduce the time required to verify a design's compliance. Furthermore, the QVT is still undergoing changes and as the QVT standardization process moves forward our QVT engine will have to evolve to comply with the operational semantics defined in the eventual QVT standard.

While we have not investigated this avenue, we feel that model transformation patterns and compliance rule patterns exist. Once such patterns have been defined the proposed prototype could help a software engineer in using them. That is, the prototype could be accompanied by a catalogue of patterns that are available to the software engineer. Being more ambitious we envision being able to guide the software engineer in selecting and using the patterns in the catalogue.

5.3 Conclusion

The arrival of the Model Driven Development paradigm has increased the use of models in the software development process. Models are now used for defining software specifications, as well as, software designs and sometimes even the implementation. The specification of a software system describes its required properties and capabilities, while the design defines in detail the actual properties and capabilities of the software system. We believe that it is important to verify that the actual properties and capabilities of the software system comply with the required

properties and capabilities. That is, it is important to verify that the design of a software system complies with the specification of the same system. The prototype proposed in this dissertation can be seen as a step towards such a reality.

We have shown that it is feasible to semi-automate the verification of a design model for compliance with a specification model using a model transformation language (QVT). This dissertation has also shown that it takes substantial effort to define the model mappings and compliance rules required for performing such verification. This observation coincides with the effort required to perform other types of verification in software engineering.

In summary, the use of QVT to enable model compliance verification has shown that the activity of model compliance verification can be semi-automated. What remains is to reduce the effort required to perform model compliance verification by increasing the degree of automation that our prototype provides.

REFERENCES

- [1] Kent Beck, "Embracing Change With Extreme Programming," *IEEE Computer*, vol. 32, pp. 70--78, 1999.
- [2] Robert V. Binder, *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
- [3] Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, vol. 21, pp. 61--72, 1988.
- [4] F. Bordeleau, "A Systematic and Traceable Progression from Scenario Models to Communicating Hierarchical State Machines," PhD Thesis, in *Department of Systems and Computer Engineering*. Ottawa, Canada: Carleton University, 1999.
- [5] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose, *Eclipse Modeling Framework: A Developer's Guide*. Addison-Wesley, 2003.
- [6] R. M. Burstall and J. Darlington, "Some Transformations for Developing Recursive Programs," presented at IEEE International Conference on Reliable Software, 1975.
- [7] R. M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs," *Journal of the ACM*, vol. 24, 1977.
- [8] Klasse Objecten : OCL Center, <http://www.klasse.nl/ocl/>, 2004.
- [9] Visio. *Microsoft Corp.* available at <http://office.microsoft.com/visio/>. 2004.
- [10] Borland Together Designer. *Borland Software Corporation* available at <http://www.borland.com/together/architect/>. 2004.
- [11] Stylus Studio 6 XML Professional Edition. *Sonic Software Corporation* available at <http://www.stylusstudio.com>. 2004.
- [12] Jean-Pierre Corriveau, "Traceability Process for Large OO Projects," *IEEE Computer*, vol. 29, pp. 63--68, 1996.
- [13] Krzysztof Czarnecki and Simon Helsen, "Classification of Model Transformation Approaches," presented at OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

- [14] J. Darlington, "An Experimental Program Transformation and Synthesis System," *Artificial Intelligence*, vol. 16, 1981.
- [15] Svata Dedic and Martin Matula, <http://java.netbeans.org/models/java/java-model.html>, 2004.
- [16] Hans-Erik Eriksson, *UML 2 Toolkit*. Indianapolis, Ind.: Wiley Pub., 2004.
- [17] David Frankel, "Model Driven Architecture: Reality and Implementation," OMG-MDA Presentation 2001.
- [18] David Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley Press, 2003.
- [19] Tracy Gardner and Catherine Griffin, "A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard," OMG document ad/03-08-02, 2002.
- [20] A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood, "Transformation: The Missing Link of MDA," presented at First International Conference Graph Transformation (ICGT), Barcelona, Spain, 2002.
- [21] Hassan Gomaa, *Designing Concurrent, Distributed, and Real-time Applications with UML*. Reading, MA: Addison-Wesley, 2000.
- [22] Andy Hunt and Dave Thomas, *Pragmatic Unit Testing: Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003.
- [23] Andy Hunt and Dave Thomas, *Pragmatic Unit Testing: Unit Testing in C# with NUnit*. The Pragmatic Programmers, 2004.
- [24] UMLAUT. *Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA)* available at <http://www.irisa.fr/UMLAUT/Welcome.html>. 2004.
- [25] International Organization for Standardization (ISO), <http://www.iso.org>, 2004.
- [26] International Telecommunications Union (ITU), <http://www.itu.org>, 2004.
- [27] Joint Tactical Radio System (JTRS) Joint Program Office (JPO), "Software Communications Architecture (SCA)."
- [28] JUnit, <http://www.junit.org>, 2004.

- [29] Anneke Kleppe, Jos Warmer, and W. Bast, *MDA Explained: The Model Driven Architecture Practice and Promise*. Addison-Wesley, 2003.
- [30] Mikael Lindvall and Kristian Sandahl, "Practical Implications of Traceability," *Software - Practice and Experience*, vol. 26, pp. 1161–1180, 1996.
- [31] Adaptive Ltd., Alcatel, CBOP, Codagen Technologies Corporation, DSTC, International Business Machines, Softbeam, TCS America, TNI-Valiosys, and Thales, "MOF Query / Views / Transformations First Revised Submission," OMG document ad/04-10-04, 2004.
- [32] Stephen J. Mellor, *MDA Distilled : Principles of Model-Driven Architecture*. Boston: Addison-Wesley, 2004.
- [33] Harlan D. Mills, Michael Dyer, and Richard Linger, "Cleanroom Software Engineering," *IEEE Software*, vol. 4, pp. 19–25, 1987.
- [34] Microsoft .NET, <http://www.microsoft.com/net/>, 2004.
- [35] NUnit, <http://www.nunit.org>, 2004.
- [36] Object Management Group (OMG), "MOF 2.0 Query / Views / Transformations RFP," OMG document ab/02-04-02, 2002.
- [37] Object Management Group (OMG), "MDA Guide v1.0.1," <http://www.omg.org/docs/omg/03-06-01.pdf>, 2003.
- [38] Object Management Group (OMG), "UML 2.0 Infrastructure Final Adopted Specification," OMG Document ptc/03-09-15, 2003.
- [39] Object Management Group (OMG), "UML 2.0 OCL Specification, Final Adopted Specification," OMG Document ptc/03-10-14 2003, 2003.
- [40] Object Management Group (OMG), "UML 2.0 Superstructure Final Adopted Specification," OMG Document ptc/03-08-02, 2003.
- [41] Object Management Group (OMG), <http://www.omg.org>, 2004.
- [42] Object Management Group (OMG), "Meta-Object Facility (MOF) 2.0," OMG Adopted Specification ptc/03-10-04, 2004.
- [43] Object Management Group (OMG), "MOF 2.0 XML Metadata Interchange (XMI)," ptc/04-06-11, 2004.

- [44] Object Management Group (OMG), "Software Radio Components Specification. Final Adopted Specification.," OMG Document dtc/04-05-05, 2004.
- [45] Object Management Group (OMG), "Unified Modeling Language (UML), Version 1.5," OMG Document formal/2003-03-01, 2004.
- [46] QVT Partners, <http://www.qvtp.org/>, 2004.
- [47] Ivan Porres, "A Toolkit for Model Manipulation," *Software and Systems Modeling*, vol. 4, 2003.
- [48] Roger S. Pressman, *Software Engineering : A Practitioner's Approach*, 6th ed. Boston, Mass.: McGraw-Hill, 2005.
- [49] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," presented at IEEE WESTCON, Los Angeles, CA, 1970.
- [50] James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, 2nd ed. Boston: Addison-Wesley, 2004.
- [51] Stephen R. Schach, *Object-Oriented and Classical Software Engineering*, 6th ed. Boston: McGraw-Hill Higher Education, 2005.
- [52] Bran Selic, "The Pragmatics of Model-Driven Development," *IEEE Software*, vol. 20, pp. 19--25, 2003.
- [53] Bran Selic, Garth Gullekson, and Paul T. Ward, *Real-time Object-Oriented Modeling*. New York: Wiley & Sons, 1994.
- [54] Bran Selic and James Rumbaugh, "Using UML for Modeling Complex Real-Time Systems," IBM <http://www-128.ibm.com/developerworks/rational/library/139.html>, 1998.
- [55] Aaron Skonnard and Martin Gudgin, *Essential XML quick reference : a programmer's reference to XML, XPath, XSLT, XML Schema, SOAP, and more*. Boston: Addison-Wesley, 2002.
- [56] IBM Rational Software, <http://www-306.ibm.com/software/rational/>, 2004.
- [57] Ian Sommerville, *Software Engineering*, 6th ed. Harlow, England ; New York: Addison-Wesley, 2000.
- [58] P. Stevens, "Small-scale XMI Programming: A Revolution in UML Tool Use?," *Automated Software Engineering*, vol. 10, 2003.

- [59] Bjarne Stroustrup, *The C++ Programming Language (Special 3rd Edition)*, 3rd ed: Addison-Wesley Professional, 2000.
- [60] Triskell Team, <http://www.irisa.fr/triskell>, 2004.
- [61] Triskell Team, "Model Transformation With a Dedicated OO Language," Triskell Team, IRISA Rennes, France 2004.
- [62] Java Technology, <http://java.sun.com/>, 2004.
- [63] Telelogic DOORS. *Telelogic* available at <http://www.telelogic.com/products/doorsers/index.cfm>. 2004.
- [64] Didier Vojtisek, "BasicMTL Realization Guide," IRISA MOTOR/004, 2004.
- [65] W3C, "XML Path (XPath) Version 1.0," <http://www.w3.org/TR/xpath> 1999.
- [66] W3C, "XSL Transformations (XSLT) Version 1.0," <http://www.w3.org/TR/xslt> 1999.
- [67] World Wide Web Consortium (W3C), "XML Schema Part 0: Primer Second Edition," 2004.
- [68] Jos Warmer and Anneke Kleppe, *The Object Constraint Language : Getting Your Models Ready for MDA*. Boston, MA: Addison-Wesley, 2003.
- [69] J. Whittle, "Transformations and Software Modeling Languages: Automating Transformations in UML," presented at UML 2002 - Model Engineering, Languages, Concepts, and Tools, Dresden, Germany, 2002.
- [70] Juan-Pablo Zamora Zapata, "Compliance Verification of a Design Model with respect to its Specification Model in the Context of Software Defined Radios: A Model Transformation Approach," in *School of Computer Science*. Ottawa: Carleton University, 2004.
- [71] Juan-Pablo Zamora Zapata, Francis Bordeleau, Jean-Pierre Corriveau, and Toby McClean, "Challenges and Possible Solutions to Model Compliance in the Context of Model Driven Development," presented at 2004 Workshop in Software Model Engineering, UML'2004, Lisbon, Portugal, 2004.

TABLE OF ACRONYMS

AUT	Artefact Under Test
CIM	Computation Independent Model
GUI	Graphical User Interface
IDE	Integrated Development Environment
MDA	Model Driven Architecture
MDD	Model Driven Development
MOF	Meta Object Facility
MUT	Mapping Under Test
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query/Views/Transformations
RFP	Request for Proposal
Rose	IBM Rational Rose
RoseRT	IBM Rational Rose Real-Time
SCA	Software Component Architecture
SRC	Software Radio Components
UML	Unified Modeling Language
XMI	XML Metadata Interchange

Appendix B

TOOL SCREEN CAPTURE

Figure 83 Prototype screen capture

Appendix C

UML AND ROSE REAL-TIME METAMODELS

In this appendix we present the UML and RoseRT[†] metamodels that we used for the Case Study presented in this dissertation.

C.1 UML metamodel

This is a copy of the metamodel presented in the UML version 1.5 specification [45].

Figure 84 Auxillary elements

Figure 86 Classifiers

Figure 87 Relationship elements

C.2RoseRT metamodel

This has been extracted from the RoseRT documentation.

Figure 88 Association elements diagram

Figure 89 RoseRT Attribute elements diagram

Figure 90 RoseRT Capsule element diagram

Figure 91 RoseRT Class elements diagram

Figure 92 RoseRT ClassifierRole elements diagram

Figure 93 RoseRT Collaboration elements diagram

Figure 94 RoseRT Connector element diagram

Figure 95 RoseRT LogicalPackage elements diagram

Figure 96 RoseRT Operation elements diagram

Figure 97 RoseRT Protocol elements diagram

Figure 98 RoseRT Relation elements diagram

Figure 99 RoseRT Classifier elements diagram

XML SCHEMAS

D.1 Model Files

This section presents the XML Schema for the prototypes model files.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xs:element name="objectmodel">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="rootdefinition"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="rootdefinition">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="objects"/>
      </xs:sequence>
      <xs:attribute name="id" use="required"/>
      <xs:attribute name="name" use="required" type="xs:NCName"/>
      <xs:attribute name="refid" use="required" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="objects">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="objectdefinition"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="objectdefinition">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="slots"/>
      </xs:sequence>
      <xs:attribute name="id" use="required"/>
      <xs:attribute name="identifier" use="required" type="xs:NCName"/>
      <xs:attribute name="name" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="slots">
```

```

    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded"
ref="slotdefinition"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="slotdefinition">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="value"/>
      </xs:sequence>
      <xs:attribute name="id" use="required"/>
      <xs:attribute name="name" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="value">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" ref="contents"/>
      </xs:sequence>
      <xs:attribute name="id" use="required"/>
      <xs:attribute name="name" use="required"/>
      <xs:attribute name="primitive" type="xs:boolean"/>
      <xs:attribute name="refid"/>
      <xs:attribute name="strval" type="xs:NCName"/>
      <xs:attribute name="type" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="contents">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="item"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="item">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" ref="objectdefinition"/>
      </xs:sequence>
      <xs:attribute name="id"/>
      <xs:attribute name="name" type="xs:NMTOKEN"/>
      <xs:attribute name="strval" type="xs:NCName"/>
      <xs:attribute name="type" use="required" type="xs:NCName"/>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

CASE STUDY : SPECIFICATION MODEL

Figure 100 Bank system specification model packages

Figure 101 Specification model BaseTypes package

Figure 102 Server and manger elements of the specification model ActiveClasses package

Figure 103 Specification model ActiveClasses PINValidationTransactionManager diagram

Figure 104 Specification model ActiveClasses QueryTransactionManager diagram

Figure 105 Specification model ActiveClasses TransferTransactionManager diagram

Figure 106 Specification model ActiveClasses WithdrawalTransactionManager diagram

Figure 107 Specification model DataClasses package

Figure 108 Specification model DataClasses package auxillary elements

Figure 109 AddInterest sequence diagram

Figure 110 CheckingToCheckingTransfer sequence diagram

Figure 111 CheckingToSavings sequence diagram

Figure 112 CheckingWithdrawFunds sequence diagram

Figure 113 CloseCheckingAccount sequence diagram

Figure 114 CloseSavingsAccount sequence diagram

Figure 115 Initialize sequence diagram

Figure 116 OpenCheckingAccount sequence diagram

Figure 117 OpenSavingsAccount sequence diagram

Figure 118 QueryCheckingAccount sequence diagram

Figure 119 QuerySavingsAccount sequence diagram

Figure 120 SavingsToCheckingTransfer sequence diagram

Figure 121 SavingsToSavingsTransfer sequence diagram

Figure 122 SavingsWithdrawFunds sequence diagram

Figure 123 TransferFundsAbort sequence diagram

Figure 124 WithdrawFundsAbort sequence diagram

CASE STUDY : DESIGN MODEL

F.1 BankSystem Package

F.1.1 Packages

- ActiveClasses
- BaseTypes
- DataClasses
- Interfaces
- SequenceDiagrams

F.1.2 Classes

There are no classes defined in this package.

F.1.3 Capsules

BANKSERVER

Structure : Ports

Name	Protocol	Configured
client	ClientInterface	false
manager	ManagementInterface	false

Structure : CapsuleRoles

Role	Type
accountManager	AccountManagementTransactionServer
checkings	CheckingAccountWrapper
debitCards	DebitCardWrapper
manager	ManagementTransactionServer
validator	PINValidationTransactionManager
queries	QueryTransactionManager
savings	SavingsAccountWrapper
log	TransactionLogWrapper
transactions	TransactionServer
transfers	TransferTransactionManager
withdrawals	WithdrawalTransactionManager

F.1.4 Protocols

There are no protocols defined in this package.

F.2ActiveClasses Package

F.2.1 Packages

There are no packages in this package.

F.2.2 Classes

There are no classes in this package.

F.2.3 Capsules

MANAGEMENTTRANSACTIONSERVER

Structure : Ports

Name	Protocol	Configured
validationManager	PINValidationTransactionManager _TO_ ManagementTransactionServer	True
withdrawalManager	WithdrawalTransactionManager _TO_ ManagementTransactionServer	True
queryManager	QueryTransactionManager _TO_ ManagementTransactionServer	True
transferManager	TransferTransactionManager _TO_ ManagementTransactionServer	True
accountManager	AccountManagementTransactionServer _TO_ ManagementTransactionServer	True
logger	TransactionLogWrapper _TO_ ManagementTransactionServer	True
to_ManagementInterface	ManagementInterface	False

WITHDRAWALTRANSACTIONMANAGER

Operations

Name	Type
confirm(accountNumber String, amount Real)	<unspecified>
abort(accountNumber String, amount Real)	<unspecified>
withdraw(accountNumber String, amount Real, accountType String)	String
initialize()	<unspecified>

Structure : Ports

Name	Protocol	Configured
manager	WithdrawalTransactionManager _TO_ ManagementTransactionServer	False
server	WithdrawalTransactionManager _TO_ TransactionServer	False
transactionLog	TransactionLogWrapper _TO_ WithdrawalTransactionManager	True
checkingAccounts	CheckingAccountWrapper _TO_ WithdrawalTransactionManager	True
savingsAccounts	SavingsAccountWrapper _TO_ WithdrawalTransactionManager	True
validationManager	PINValidationTransactionManager _TO_ WithdrawalTransactionManager	True
thePINValidationTransactionManager	PINValidationTransactionManager _TO_ WithdrawalTransactionManager	True

TRANSFER TRANSACTION MANAGER

Operations

Name	Type
confirm(accountNumber String, amount Double)	<unspecified>
abort(accountNumber String, amount Double)	<unspecified>
initialize()	<unspecified>
transfer(fromAccountNumber String, toAccountNumber String, amount Real, fromAccountType String, toAccountType String)	String

Structure : Ports

Name	Protocol	Configured
manager	TransferTransactionManager _TO_ ManagementTransactionServer	False
server	TransferTransactionManager _TO_ TransactionServer	False
savingsAccounts	SavingsAccountWrapper _TO_ TransferTransactionManager	True
checkingAccounts	CheckingAccountWrapper _TO_ TransferTransactionManager	True
transactionsLog	TransactionLogWrapper _TO_ TransferTransactionManager	True
validationManager	PINValidationTransactionManager _TO_ TransferTransactionManager	True

TRANSACTIONSERVER

Operations

Name	Type
getConfirmation(accountNumber)	<unspecified>

Structure : Ports

Name	Protocol	Configured
transferManager	TransferTransactionManager _TO_ TransactionServer	True
queryManager	QueryTransactionManager _TO_ TransactionServer	True
withdrawalManager	WithdrawalTransactionManager _TO_ TransactionServer	True
validationManager	PINValidationTransactionManager _TO_ TransactionServer	True
to_ClientInterface	ClientInterface	False

TRANSACTIONLOGWRAPPER

Operations

Signal	Type
log(transaction Transaction)	<unspecified>
read()	Transaction

Structure : Ports

Name	Protocol	Configured
accountManager	TransactionLogWrapper _TO_ AccountManagementTransactionServer	False
withdrawalManager	TransactionLogWrapper _TO_ WithdrawalTransactionManager	False
transferManager	TransactionLogWrapper _TO_ TransferTransactionManager	False
queryManager	TransactionLogWrapper _TO_ QueryTransactionManager	False
manager	TransactionLogWrapper _TO_ ManagementTransactionServer	False

SAVINGSACCOUNTWRAPPER

Operations

Signal	Type
credit(accountNumber String, amount Real)	<unspecified>
open(accountNumber String)	<unspecified>
readBalance(accountNumber String)	Real
close(accountNumber String)	<unspecified>
readCumulativeInterest(accountNumber String)	Real
debit(accountNumber String, amount Real)	<unspecified>
addInterest(accountNumber String, interestRate Real)	<unspecified>
readLastDepositAmount(accountNumber String)	Real

Structure : Ports

Name	Protocol	Configured
accountManager	SavingsAccountWrapper _TO_ AccountManagementTransactionServer	False
withdrawalManager	SavingsAccountWrapper _TO_ WithdrawalTransactionManager	False
queryManager	SavingsAccountWrapper _TO_ QueryTransactionManager	False
transferManager	SavingsAccountWrapper _TO_ TransferTransactionManager	False

QUERYTRANSACTIONMANAGER

Operations

Signature	Type
query(accountNumber String)	String
initialize()	<unspecified>

Structure : Ports

Name	Protocol	Configured
manager	QueryTransactionManager _TO_ ManagementTransactionServer	False
server	QueryTransactionManager _TO_ TransactionServer	False
savingsAccounts	SavingsAccountWrapper _TO_ QueryTransactionManager	True
checkingAccounts	CheckingAccountWrapper _TO_ QueryTransactionManager	True
transactionLog	TransactionLogWrapper _TO_ QueryTransactionManager	True

PINValidationTransactionManager**Operations**

Signature	Type
validatePIN(cardId String, PIN String)	String
initialize()	<unspecified>

Structure : Ports

Name	Protocol	Configured
manager	PINValidationTransactionManager _TO_ ManagementTransactionServer	False
server	PINValidationTransactionManager _TO_ TransactionServer	False
withdrawalManager	PINValidationTransactionManager _TO_ WithdrawalTransactionManager	False
transferManager	PINValidationTransactionManager _TO_ TransferTransactionManager	False
theWithdrawalTransactionManager	PINValidationTransactionManager _TO_ WithdrawalTransactionManager	False
debitCards	DebitCardWrapper _TO_ PINValidationTransactionManager	True

DebitCardWrapper**Operations**

Signature	Type
delete(cardId String)	<unspecified>
clearTotal(cardId String)	<unspecified>
updateExpirationDate(cardId String, expirationDate Date)	<unspecified>
updateDailyTotal(cardId String, amount Real)	<unspecified>
updateCardStatus(cardId String, status CardStatus)	<unspecified>
validate(cardId String, PIN String)	<unspecified>
checkDailyLimit(cardId String, amount Real)	<unspecified>
read(cardId String)	CardInfo
updateDailyLimit(cardId String, newLimit Real)	<unspecified>
updatePIN(cardId String, PIN String)	<unspecified>
create(cardId String)	<unspecified>

Structure : Ports

Name	Protocol	Configured
validationManager	DebitCardWrapper _TO_ PINValidationTransactionManager	False

CHECKINGACCOUNTWRAPPER

Operations

Signature	Type
close(accountNumber String)	<unspecified>
readLastDepositAmount(accountNumber String)	Real
credit(accountNumber String, amount Real)	<unspecified>
readBalance(accountNumber String)	Real
open(accountNumber String)	<unspecified>
debit(accountNumber String, amount Real)	<unspecified>

Structure : Ports

Name	Protocol	Configured
withdrawalManager	CheckingAccountWrapper _TO_ WithdrawalTransactionManager	False
queryManager	CheckingAccountWrapper _TO_ QueryTransactionManager	False
transferManager	CheckingAccountWrapper _TO_ TransferTransactionManager	False
accountManager	CheckingAccountWrapper _TO_ AccountManagementTransactionServer	False

ACCOUNTMANAGEMENTTRANSACTIONSERVER

Operations

Signature	Type
close(accountNumber String)	<unspecified>
open(accountNumber String)	<unspecified>
addInterest(accountNumber String, interestRate Real)	<unspecified>

Structure : Ports

Name	Protocol	Configured
manager	AccountManagementTransactionServer _TO_ ManagementTransactionServer	False
logger	TransactionLogWrapper _TO_ AccountManagementTransactionServer	True
savingsAccounts	SavingsAccountWrapper _TO_ AccountManagementTransactionServer	True
checkingAccounts	CheckingAccountWrapper _TO_ AccountManagementTransactionServer	True

F.2.4 Protocols

TRANSACTIONLOGWRAPPER_TO_ACCOUNTMANAGEMENTTRANSACTIONSERVER

In Signals

Name	Type
log	logDATA_CLASS_FROM

Out Signals

Name	Type
log_RESULT	void

SAVINGSACCOUNTWRAPPER_TO_ACCOUNTMANAGEMENTTRANSACTIONSERVER

In Signals

Name	Type
open	openDATA_CLASS_FROM
close	closeDATA_CLASS_FROM
addInterest	addInterestDATA_CLASS_FROM

Out Signals

Name	Type
open_RESULT	void
close_RESULT	void
addInterest_RESULT	void

PINVALIDATIONTRANSACTIONMANAGER_TO_MANAGEMENTTRANSACTIONSERVER

In Signals

Name	Type
initialize	void

Out Signals

Name	Type
initialize_RESULT	void

WITHDRAWALTRANSACTIONMANAGER_TO_MANAGEMENTTRANSACTIONSERVER

In Signals

Name	Type
initialize	void

Out Signals

Name	Type
initialize_RESULT	void

QUERYTRANSACTIONMANAGER_TO_MANAGEMENTTRANSACTIONSERVER**In Signals**

Name	Type
initialize	void

Out Signals

Name	Type
initialize_RESULT	void

TRANSFERTRANSACTIONMANAGER_TO_MANAGEMENTTRANSACTIONSERVER**In Signals**

Name	Type
initialize	void

Out Signals

Name	Type
initialize_RESULT	void

DEBITCARDWRAPPER_TO_PINVALIDATIONTRANSACTIONMANAGER**In Signals**

Name	Type
validate	validateDATA_CLASS_FROM

Out Signals

Name	Type
validate_RESULT	void

TRANSACTIONLOGWRAPPER_TO_WITHDRAWALTRANSACTIONMANAGER**In Signals**

Name	Type
log	logDATA_CLASS_FROM

Out Signals

Name	Type
log_RESULT	void

CHECKINGACCOUNTWRAPPER_TO_WITHDRAWALTRANSACTIONMANAGER**In Signals**

Name	Type
debit	debitDATA_CLASS_FROM

Out Signals

Name	Type
debit_RESULT	void

SAVINGSACCOUNTWRAPPER_TO_WITHDRAWALTRANSACTIONMANAGER**In Signals**

There are no in signals for this Protocol.

Out Signals

There are no out signals for this Protocol.

SAVINGSACCOUNTWRAPPER_TO_QUERYTRANSACTIONMANAGER**In Signals**

Name	Type
readBalance	readBalanceDATA_CLASS_FROM
readLastDepositAmount	readLastDepositAmountDATA_CLASS_FROM
readCumulativeInterest	readCumulativeInterestDATA_CLASS_FROM

Out Signals

Name	Type
readBalance_RESULT	Real
readLastDepositAmount_RESULT	Real
readCumulativeInterest_RESULT	Real

SAVINGSACCOUNTWRAPPER_TO_TRANSFERTRANSACTIONMANAGER**In Signals**

Name	Type
debit	debitDATA_CLASS_FROM
credit	creditDATA_CLASS_FROM

Out Signals

Name	Type
debit_RESULT	void
credit_RESULT	void

CHECKINGACCOUNTWRAPPER_TO_QUERYTRANSACTIONMANAGER**In Signals**

Name	Type
readBalance	readBalanceDATA_CLASS_FROM
readLastDepositAmount	readLastDepositAmountDATA_CLASS_FROM

Out Signals

Name	Type
readBalance_RESULT	Real
readLastDepositAmount_RESULT	Real

TRANSFERTRANSACTIONMANAGER_TO_TRANSACTIONSERVER**In Signals**

Name	Type
transfer	transferDATA_CLASS_FROM
getConfirmation_RESULT	void
confirm	confirmDATA_CLASS_FROM
abort	abortDATA_CLASS_FROM

Out Signals

Name	Type
transfer_RESULT	String
getConfirmation	getConfirmationDATA_CLASS_FROM
confirm_RESULT	void
abort_RESULT	void

QUERYTRANSACTIONMANAGER_TO_TRANSACTIONSERVER**In Signals**

Name	Type
query	queryDATA_CLASS_FROM

Out Signals

Name	Type
query_RESULT	String

WITHDRAWALTRANSACTIONMANAGER_TO_TRANSACTIONSERVER**In Signals**

Name	Type
withdraw	withdrawDATA_CLASS_FROM
getConfirmation_RESULT	void
confirm	confirmDATA_CLASS_FROM
abort	abortDATA_CLASS_FROM

Out Signals

Name	Type
withdraw_RESULT	String
getConfirmation	getConfirmationDATA_CLASS_FROM
confirm_RESULT	void
abort_RESULT	void

PINVALIDATIONTRANSACTIONMANAGER_TO_TRANSACTIONSERVER**In Signals**

Name	Type
validatePIN	validatePINDATA_CLASS_FROM

Out Signals

Name	Type
validatePIN_RESULT	String

CHECKINGACCOUNTWRAPPER_TO_TRANSFERTRANSACTIONMANAGER**In Signals**

Name	Type
debit	debitDATA_CLASS_FROM
credit	creditDATA_CLASS_FROM

Out Signals

Name	Type
debit_RESULT	void
credit_RESULT	void

TRANSACTIONLOGWRAPPER_TO_TRANSFERTRANSACTIONMANAGER**In Signals**

Name	Type
log	logDATA_CLASS_FROM

Out Signals

Name	Type
log_RESULT	void

TRANSACTIONLOGWRAPPER_TO_QUERYTRANSACTIONMANAGER**In Signals**

Name	Type
read	void
log	logDATA_CLASS_FROM

Out Signals

Name	Type
read_RESULT	Transaction
log_RESULT	void

ACCOUNTMANAGEMENTTRANSACTIONSERVER_TO_MANAGEMENTTRANSACTIONSERVER**In Signals**

Name	Type
open	openDATA_CLASS_FROM
close	closeDATA_CLASS_FROM
addInterest	addInterestDATA_CLASS_FROM

Out Signals

Name	Type
open_RESULT	void
close_RESULT	void
addInterest_RESULT	void

CHECKINGACCOUNTWRAPPER_TO_ACCOUNTMANAGEMENTTRANSACTIONSERVER**In Signals**

Name	Type
open	openDATA_CLASS_FROM
close	closeDATA_CLASS_FROM

Out Signals

Name	Type
open_RESULT	void
close_RESULT	void

PINVALIDATIONTRANSACTIONMANAGER_TO_WITHDRAWALTRANSACTIONMANAGER**In Signals**

Name	Type
validatePIN	validatePINDATA_CLASS_FROM

Out Signals

Name	Type
validatePIN_RESULT	String

PINVALIDATIONTRANSACTIONMANAGER_TO_TRANSFERTRANSACTIONMANAGER**In Signals**

Name	Type
validatePIN	validatePINDATA_CLASS_FROM

Out Signals

Name	Type
validatePIN_RESULT	String

TRANSACTIONLOGWRAPPER_TO_MANAGEMENTTRANSACTIONSERVER**In Signals**

Name	Type
log	logDATA_CLASS_FROM

Out Signals

Name	Type
log_RESULT	void

F.2.5 Associations

There are no associations in this package.

F.3BaseTypes Package

F.3.1 Packages

There are no packages in this package.

F.3.2 Classes

STRING

Generalizations

Supplier name	PrimitiveType

REAL

Generalizations

Supplier name	PrimitiveType

INTEGER

Generalizations

Supplier name	PrimitiveType

BOOLEAN

Generalizations

Supplier name	PrimitiveType

SET

Generalizations

Supplier name	Collection

SEQUENCE

Generalizations

Supplier name	Collection

ORDEREDSET

Generalizations

Supplier name	Collection

BAG**DATE****COLLECTION****PRIMITIVE TYPE****F.3.3 Capsules**

There are no capsules in this package.

F.3.4 Protocols

There are no protocols in this package.

F.3.5 Associations

There are no associations in this package.

F.4 DataClasses Package**F.4.1 Packages**

There are no packages in this package.

F.4.2 Classes

ACCOUNT

Attributes

Name	Type
accountNumber	String
balance	Real

BANK

Attributes

Name	Type
name	String

CARDINFO

Attributes

Name	Type
PIN	String
expirationDate	Date
limit	Real
accountNumber	String
status	CardStatus
total	Real

CARDSTATUS

Attributes

Name	Type
isActive	Boolean

CHECKINGACCOUNT

Generalizations

Supplier name	Account

CUSTOMER

Generalizations

Supplier name	Person

Attributes

Name	Type
creditRating	Integer
occupation	String

SAVINGSACCOUNT

Generalizations

Supplier name	Account

Attributes

Name	Type
accumulatedInterest	Real

Operations

Signature	Type
calculateInterest(rate Real)	Real
addInterest(interest Real)	<unspecified>

TRANSACTION

Attributes

Name	Type
amount	Real
type	String
date	Date
message	String
accountNumber	String

ADDRESS

Attributes

Name	Type
province	String
postalCode	String
street	String

EMPLOYEE

Generalizations

Supplier name	Person

Attributes

Name	Type
employeeNumber	String
startDate	Date
salary	Real

Operations

Signature	Type
yearsOfService()	Integer

MANAGER

Generalizations

Supplier name	Employee

BANKBRANCH

PERSON

Attributes

Name	Type
dob	Date
firstName	String
lastName	String

Operations

Signature	Return Type
age()	Integer

F.4.3 Capsules

There are no capsules in this package.

F.4.4 Protocols

There are no protocols in this package.

F.4.5 Associations**ASSOCIATION**

end	Name	Classifier	Aggregate	Containment
end1	account	Account	False	unspecified
end2	owner	Customer	False	unspecified

ASSOCIATION

end	Name	Classifier	Aggregate	Containment
end1	clients	Customer	False	unspecified
end2	bank	Bank	False	unspecified

ASSOCIATION

end	Name	Classifier	Aggregate	Containment
end1	reportees	Employee	False	unspecified
end2	reportTo	Manager	False	unspecified

ASSOCIATION

end	Name	Classifier	Aggregate	Containment
end1	manager	Manager	False	unspecified
end2	branch	BankBranch	False	unspecified

ASSOCIATION

end	Name	Classifier	Aggregate	Containment
end1	transit	BankBranch	False	unspecified
end2	account	Account	False	unspecified

ASSOCIATION

end	Name	Classifier	Aggregate	Containment
end1	address	Address	False	Ref
end2	branch	BankBranch	True	unspecified

ASSOCIATION

end	Name	Classifier	Aggregate	Containment
end1	branches	BankBranch	False	unspecified
end2	parent	Bank	False	unspecified

ASSOCIATION

end	Name	Classifier	Aggregate	Containment
end1	worksAt	BankBranch	False	unspecified
end2	employee	Employee	False	unspecified

ASSOCIATION

end	Name	Classifier	Aggregate	Containment
end1	address	Address	False	Ref
end2	thePerson	Person	True	unspecified

F.5 Interfaces Package

F.5.1 Packages

There are no packages in this package.

F.5.2 Classes

OPENDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String

ADDINTERESTDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String
interestRate	Real

CLOSEDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String

READCUMULATIVEINTERESTDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String

WITHDRAWDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String
amount	Real
accountType	String

READLASTDEPOSITAMOUNTDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String

ABORTDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String
amount	Real

CONFIRMDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String
amount	Real

TRANSFERDATA_CLASS_FROM

Attributes

Name	Type
fromAccountNumber	String
toAccountNumber	String
amount	Real
fromAccountType	String
toAccountType	String

READBALANCEDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String

F.5.3 Capsules

There are no capsules in this package.

F.5.4 Protocols**MANAGEMENTINTERFACE**

In Signals

Name	Type
open	openDATA_CLASS_FROM
initialize	void
addInterest	addInterestDATA_CLASS_FROM
close	closeDATA_CLASS_FROM

Out Signals

Name	Type
open_RESULT	void
initialize_RESULT	void
addInterest_RESULT	void
close_RESULT	void

CLIENTINTERFACE**In Signals**

Name	Type
recentTransactions	void
readCumulativeInterest	readCumulativeInterestDATA_CLASS_FROM
withdraw	withdrawDATA_CLASS_FROM
readLastDepositAmount	readLastDepositAmountDATA_CLASS_FROM
abort	abortDATA_CLASS_FROM
confirm	confirmDATA_CLASS_FROM
transfer	transferDATA_CLASS_FROM
readBalance	readBalanceDATA_CLASS_FROM

Out Signals

Name	Type
recentTransactions_RESULT	Transaction
readCumulativeInterest_RESULT	Real
withdraw_RESULT	String
readLastDepositAmount_RESULT	Real
abort_RESULT	void
confirm_RESULT	void
transfer_RESULT	String
readBalance_RESULT	Real

F.5.5 Associations

There are no associations in this package.

F.6SequenceDiagrams Package**F.6.1 Packages**

There are no packages in this package.

F.6.2 Classes

WITHDRAWDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String
amount	Real
accountType	String

VALIDATEPINDATA_CLASS_FROM

Attributes

Name	Type
cardId	String
PIN	String

GETCONFIRMATIONDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	

CONFIRMDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String
amount	Real

VALIDATEDATA_CLASS_FROM

Attributes

Name	Type
cardId	String
PIN	String

DEBITDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String
amount	Real

LOGDATA_CLASS_FROM

Attributes

Name	Type
transaction	Transaction

TRANSFERDATA_CLASS_FROM

Attributes

Name	Type
fromAccountNumber	String
toAccountNumber	String
amount	Real
fromAccountType	String
toAccountType	String

CREDITDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String
amount	Real

ABORTDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String
amount	Real

QUERYDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String

READBALANCEDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String

READLASTDEPOSITAMOUNTDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String

READCUMULATIVEINTERESTDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String

OPENDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String

CLOSEDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String

ADDINTERESTDATA_CLASS_FROM

Attributes

Name	Type
accountNumber	String
interestRate	Real

F.6.3 Capsules

There are no capsules in this package.

F.6.4 Protocols

There are no protocols in this package.

F.6.5 Associations

There are no associations in this package.

CASE STUDY : TRACEABILITY MODEL EXTRACTS

This Appendix contains excerpts from the traceability model produced for the case study from Chapter 4. For each of the mappings defined in the case study an example of the traceability information produced is provided.

G.1 UMLToRoseRT

```
<trace>
  <mapping-applied>UMLToRoseRT</mapping-applied>
  <source-objects>
    <obj type="Package" valId="92e175f3-b04b-4918-b88a-54b2b37e2884" name="Model"/>
    <obj type="AdditionalInfo" valId="53caf3b8-cea1-434d-9255-e009af7ba527" name=""/>
  </source-objects>
  <target-objects>
    <obj type="LogicalPackage" valId="bae0022a-aaad-428a-b000-fcd53ff07317" name="Model"/>
  </target-objects>
</trace>
```

G.2 ProcessInteractions

```
<trace>
  <mapping-applied>ProcessInteractions</mapping-applied>
  <source-objects>
    <obj type="Package" valId="2f9cbb5b-97ed-4da2-83dc-6f5ae23e09ba" name="BaseTypes"/>
  </source-objects>
  <target-objects/>
</trace>
```

G.3 ParameterToParameter

```
<trace>
  <mapping-applied>ParameterToParameter</mapping-applied>
  <source-objects>
    <obj type="Parameter" valId="535b4f10-be13-4da4-b93b-2f9493208ca9" name="amount"/>
  </source-objects>
  <target-objects>
    <obj type="Parameter" valId="1701b379-4667-42b0-91be-11530d64b9d3" name="amount"/>
  </target-objects>
</trace>
```

```

</target-objects>
</trace>

```

G.4 ParameterToAttribute

```

<trace>
  <mapping-applied>ParameterToAttribute</mapping-applied>
  <source-objects>
    <obj type="Parameter" valId="dc0bfcbd-f959-44bb-b80a-39d049f9d6f7" name="PIN"/>
  </source-objects>
  <target-objects>
    <obj type="Attribute" valId="2816e783-cd13-4534-a054-28908765b485" name="PIN"/>
  </target-objects>
</trace>

```

G.5 PackageToLogicalPackage

```

<trace>
  <mapping-applied>PackageToLogicalPackage</mapping-applied>
  <source-objects>
    <obj type="Package" valId="75ed68f1-ba3a-49dd-9893-c1c618a27210" name="ActiveClasses"/>
    <obj type="AdditionalInfo" valId="53caf3b8-cea1-434d-9255-e009af7ba527" name=""/>
  </source-objects>
  <target-objects>
    <obj type="LogicalPackage" valId="b65012ae-6d2f-4b9b-a71b-336add3fe920" name="ActiveClasses"/>
  </target-objects>
</trace>

```

G.6 OperationToSignal

```

<trace>
  <mapping-applied>OperationToSignal</mapping-applied>
  <source-objects>
    <obj type="Operation" valId="41901A6C01FD" name="abort"/>
  </source-objects>
  <target-objects>
    <obj type="Signal" valId="18fd3bb4-9fb8-4d06-86e4-456be1d7bf99" name="abort"/>
  </target-objects>
</trace>

```

G.7 OperationToOperation

```

<trace>
  <mapping-applied>OperationToOperation</mapping-applied>
  <source-objects>
    <obj type="Operation" valId="41901F6301F0" name="open"/>
  </source-objects>

```

```

    <target-objects>
      <obj type="Operation" valId="9abfd638-661f-4384-b8de-a99c54ce467a" name="open" />
    </target-objects>
  </trace>

```

G.8 OperationToDataClass

```

<trace>
  <mapping-applied>OperationToDataClass</mapping-applied>
  <source-objects>
    <obj type="Operation" valId="41901EB10050" name="close" />
  </source-objects>
  <target-objects>
    <obj type="Class" valId="c0a52216-31e7-4306-8936-ad7096fdfa14" name="closeDATA_CLASS_FROM" />
  </target-objects>
</trace>

```

G.9 OperationResultToSignal

```

<trace>
  <mapping-applied>OperationResultToSignal</mapping-applied>
  <source-objects>
    <obj type="Operation" valId="41901EB10050" name="close" />
  </source-objects>
  <target-objects>
    <obj type="Signal" valId="25cc8331-3e6d-4ce9-8643-a2ac6a589d1b" name="close_RESULT" />
  </target-objects>
</trace>

```

G.10 MessageToSignal

```

<trace>
  <mapping-applied>MessageToSignal</mapping-applied>
  <source-objects>
    <obj type="Message" valId="4197F45F0270" name="log()" />
  </source-objects>
  <target-objects>
    <obj type="Signal" valId="2dc36e9c-3a87-48ef-bcf2-6f863d7b1b77" name="log" />
  </target-objects>
</trace>

```

G.11 InterfaceToProtocol

```

<trace>
  <mapping-applied>InterfaceToProtocol</mapping-applied>
  <source-objects>
    <obj type="Interface" valId="4193FC1C03C7" name="ClientInterface" />
  </source-objects>
  <target-objects>
    <obj type="Protocol" valId="4193FC1C03C7" name="ClientInterface" />
  </target-objects>
</trace>

```



```

</source-objects>
<target-objects>
  <obj type="Protocol" valId="55fde5cf-d846-42db-a7f3-14d9a5d6681e" name="ClientInterface"/>
</target-objects>
</trace>

```

G.12 InterfaceToPortName

```

<trace>
  <mapping-applied>InterfaceToPortName</mapping-applied>
  <source-objects>
    <obj type="Interface" valId="4193FC1C03C7" name="ClientInterface"/>
  </source-objects>
  <target-objects>
    <obj type="SUCCESS" valId="46389c44-2bb7-4ecc-9cbf-a3788d13a6f4" name=""/>
  </target-objects>
</trace>

```

G.13 InterfaceToDataClasses

```

<trace>
  <mapping-applied>InterfaceToDataClasses</mapping-applied>
  <source-objects>
    <obj type="Interface" valId="4197FADC0224" name="ManagementInterface"/>
  </source-objects>
  <target-objects>
    <obj type="Class" valId="e0ecf0e2-7b46-4fd9-8fcb-34ecffbfcf6" name="openDATA_CLASS_FROM"/>
    <obj type="Class" valId="decede90-be35-4b1f-927f-c4e4c7f38323"
name="addInterestDATA_CLASS_FROM"/>
    <obj type="Class" valId="52bc2bd8-35d5-4d11-b498-cb7eecd66fab" name="closeDATA_CLASS_FROM"/>
  </target-objects>
</trace>

```

G.14 InterfaceOperationToSignal

```

<trace>
  <mapping-applied>InterfaceOperationToSignal</mapping-applied>
  <source-objects>
    <obj type="Operation" valId="4193FB630277" name="readBalance"/>
  </source-objects>
  <target-objects>
    <obj type="Signal" valId="c29dd7fd-dd5c-4c96-bf2d-34ff90b5d68f" name="readBalance"/>
  </target-objects>
</trace>

```

G.15 InteractionToDataClasses

```

<trace>
  <mapping-applied>InteractionToDataClasses</mapping-applied>
  <source-objects>
    <obj type="Interaction" valId="e4930edb-0865-4e39-8a7f-929dbc3d2f42" name="AddInterest"/>
  </source-objects>
  <target-objects>
    <obj type="Class" valId="19bed3a4-2312-4bf5-87e2-f78e21c7d58d"
name="addInterestDATA_CLASS_FROM"/>
    <obj type="Class" valId="e3d12869-67ff-4448-b5ce-16dd038307c2"
name="addInterestDATA_CLASS_FROM"/>
    <obj type="Class" valId="ac606e88-5789-447a-8c31-e22848953766" name="logDATA_CLASS_FROM"/>
  </target-objects>
</trace>

```

G.16 InteractionAddToProtocol

```

<trace>
  <mapping-applied>InteractionAddToProtocol</mapping-applied>
  <source-objects>
    <obj type="Interaction" valId="76362ea0-2f2c-4a08-b9c3-84b00cf52e92" name="CloseSavingsAccount"/>
  </source-objects>
  <target-objects>
    <obj type="StrVal" valId="0721b8b0-80c1-42f2-8022-2dd1636e5452" name=""/>
    <obj type="Set" valId="b349bf3c-ff52-4e3d-8e28-f5869bd46705" name=""/>
  </target-objects>
</trace>

```

G.17 GeneralizationToPort

```

<trace>
  <mapping-applied>GeneralizationToPort</mapping-applied>
  <source-objects>
    <obj type="Generalization" valId="b4939c09-7fde-4ad8-9053-0f9b78631256" name=""/>
  </source-objects>
  <target-objects>
    <obj type="Port" valId="b1224197-63c5-4a5d-95a5-aeb5a90d873a" name="to_ManagementInterface"/>
  </target-objects>
</trace>

```

G.18 GeneralizationToGeneralization

```

<trace>
  <mapping-applied>GeneralizationToGeneralization</mapping-applied>
  <source-objects>
    <obj type="Generalization" valId="4f4eca7e-610b-496a-bfa9-56becabd81d" name=""/>
  </source-objects>

```

```

<target-objects>
  <obj type="Generalization" valId="c4769354-65ab-4006-90ac-45d955556b3a" name="" />
</target-objects>
</trace>

```

G.19 CollectProtocols

```

<trace>
  <mapping-applied>CollectProtocols</mapping-applied>
  <source-objects>
    <obj type="LogicalPackage" valId="bae0022a-aaad-428a-b000-fcd53ff07317" name="Model" />
  </source-objects>
  <target-objects>
    <obj type="Protocol" valId="cc9c079b-7617-4dab-b87e-1791f9883f5c"
name="TransactionLogWrapper_TO_AccountManagementTransactionServer" />
    <obj type="Protocol" valId="c79ceb90-ab11-4c72-a8e4-3140c83092f2"
name="SavingsAccountWrapper_TO_AccountManagementTransactionServer" />
    <obj type="Protocol" valId="7536999f-ce14-4c38-93ff-c9aa2a9b3196"
name="PINValidationTransactionManager_TO_ManagementTransactionServer" />
    <obj type="Protocol" valId="8a1f7374-12e7-4502-b05a-f20897954a3d"
name="WithdrawalTransactionManager_TO_ManagementTransactionServer" />
    <obj type="Protocol" valId="f10054cb-f769-40e1-87ca-93306b409fee"
name="QueryTransactionManager_TO_ManagementTransactionServer" />
    <obj type="Protocol" valId="b2f69f73-737f-4bc3-8bfb-a5d727aef995"
name="TransferTransactionManager_TO_ManagementTransactionServer" />
    <obj type="Protocol" valId="98e7edc7-f5bd-475d-a9e8-464509f39ebb"
name="DebitCardWrapper_TO_PINValidationTransactionManager" />
    <obj type="Protocol" valId="4614967b-6b4c-4a40-97e2-bc6a5a6e42dc"
name="TransactionLogWrapper_TO_WithdrawalTransactionManager" />
    <obj type="Protocol" valId="a8cdfb40-43d1-4084-91d9-9aff8e4011b1"
name="CheckingAccountWrapper_TO_WithdrawalTransactionManager" />
    <obj type="Protocol" valId="8a952dde-8b0e-4fb7-b38a-6b28fb82c339"
name="SavingsAccountWrapper_TO_WithdrawalTransactionManager" />
    <obj type="Protocol" valId="71548d98-6b5b-4310-85dc-f9e28e9d7a2"
name="SavingsAccountWrapper_TO_QueryTransactionManager" />
    <obj type="Protocol" valId="bbe7fbb5-86b4-4b95-9424-4cf547688201"
name="SavingsAccountWrapper_TO_TransferTransactionManager" />
    <obj type="Protocol" valId="e8e3ddb9-a010-4a3b-a115-09e44b4f1371"
name="CheckingAccountWrapper_TO_QueryTransactionManager" />
    <obj type="Protocol" valId="fec51bb7-df70-42d7-abbb-bd29f1983cd7"
name="TransferTransactionManager_TO_TransactionServer" />
    <obj type="Protocol" valId="460bd8ed-64f4-4a52-ba1a-2d48ac8f2349"
name="QueryTransactionManager_TO_TransactionServer" />
    <obj type="Protocol" valId="3d5bbc00-6a12-4f59-bec8-5cc67207c6f5"
name="WithdrawalTransactionManager_TO_TransactionServer" />
    <obj type="Protocol" valId="e3c55743-7162-4539-ba5d-5a7970873f62"
name="PINValidationTransactionManager_TO_TransactionServer" />
    <obj type="Protocol" valId="f641d2fc-5c6d-43db-8896-73915e475ad6"
name="CheckingAccountWrapper_TO_TransferTransactionManager" />
    <obj type="Protocol" valId="f36391d2-6191-477d-85f6-12078959ee9c"
name="TransactionLogWrapper_TO_TransferTransactionManager" />
    <obj type="Protocol" valId="8f6fa139-c098-41e2-8487-cbf264032b4e"
name="TransactionLogWrapper_TO_QueryTransactionManager" />
  </target-objects>
</trace>

```

```

    <obj type="Protocol" valId="7e81fbff-03fd-43ca-b85c-31339c5308e2"
name="AccountManagementTransactionServer_TO_ManagementTransactionServer"/>
    <obj type="Protocol" valId="9076109b-14d9-4124-8230-543682b15ca6"
name="CheckingAccountWrapper_TO_AccountManagementTransactionServer"/>
    <obj type="Protocol" valId="914f70cd-4108-49d2-9425-7c57bbc9f098"
name="PINValidationTransactionManager_TO_WithdrawalTransactionManager"/>
    <obj type="Protocol" valId="2e64b39a-196d-4582-9058-b772b5d39f36"
name="PINValidationTransactionManager_TO_TransferTransactionManager"/>
    <obj type="Protocol" valId="2f3921bc-3aa5-40ea-b52a-403002c2ba2a"
name="TransactionLogWrapper_TO_ManagementTransactionServer"/>
    <obj type="Protocol" valId="0a734d6a-c7f0-43e0-991f-c1077a63fece"
name="PINValidationTransactionManager_TO_WithdrawalTransactionManager"/>
    <obj type="Protocol" valId="ebddd7d8-75d0-4e93-ba03-427b85ab1d0f" name="ManagementInterface"/>
    <obj type="Protocol" valId="55fde5cf-d846-42db-a7f3-14d9a5d6681e" name="ClientInterface"/>
  </target-objects>
</trace>

```

G.20 ClassToDataClass

```

<trace>
  <mapping-applied>ClassToDataClass</mapping-applied>
  <source-objects>
    <obj type="Class" valId="4197FBEF007B" name="Set"/>
  </source-objects>
  <target-objects>
    <obj type="Class" valId="4197FBEF007B" name="Set"/>
  </target-objects>
</trace>

```

G.21 ClassToCapsule

```

<trace>
  <mapping-applied>ClassToCapsule</mapping-applied>
  <source-objects>
    <obj type="Class" valId="4190169402D4" name="TransactionServer"/>
  </source-objects>
  <target-objects>
    <obj type="Capsule" valId="4190169402D4" name="TransactionServer"/>
  </target-objects>
</trace>

```

G.22 AttributeToAttribute

```

<trace>
  <mapping-applied>AttributeToAttribute</mapping-applied>
  <source-objects>
    <obj type="Attribute" valId="aff6d7f7-0e00-49f4-b849-b80f3ffcb1fb" name="lastName"/>
  </source-objects>
  <target-objects>
    <obj type="Attribute" valId="eb92a412-8fe4-4cd6-b1c7-977106b45b5b" name="lastNName"/>
  </target-objects>
</trace>

```

```

</target-objects>
</trace>

```

G.23 AssociationToProtocol

```

<trace>
  <mapping-applied>AssociationToProtocol</mapping-applied>
  <source-objects>
    <obj type="Association" valId="6a9e4009-028a-41bf-9502-7254207dd612" name=""/>
  </source-objects>
  <target-objects>
    <obj type="Protocol" valId="460bd8ed-64f4-4a52-ba1a-2d48ac8f2349"
name="QueryTransactionManager_TO_TransactionServer"/>
  </target-objects>
</trace>

```

G.24 AssociationToPort

```

<trace>
  <mapping-applied>AssociationToPort</mapping-applied>
  <source-objects>
    <obj type="Association" valId="4a1b5b2e-2496-4d9a-993e-72c812ea67da" name=""/>
  </source-objects>
  <target-objects>
    <obj type="Port" valId="0ebd27b7-0dd3-4c92-89a2-2ea4759702cc" name="savingsAccounts"/>
  </target-objects>
</trace>

```

G.25 AssociationToAssociation

```

<trace>
  <mapping-applied>AssociationToAssociation</mapping-applied>
  <source-objects>
    <obj type="Association" valId="66a508c0-44fa-47f1-a15e-e23fbfea4180" name=""/>
  </source-objects>
  <target-objects>
    <obj type="Association" valId="63cd1819-0716-44bf-947c-84f52bde33cb" name=""/>
  </target-objects>
</trace>

```

G.26 AssociationEndsToProtocolName

```

<trace>
  <mapping-applied>AssociationEndsToProtocolName</mapping-applied>
  <source-objects>
    <obj type="AssociationEnd" valId="68ddddd-37b8-4034-bcde-bda96a932516" name="savingsAccounts"/>
    <obj type="AssociationEnd" valId="3006e6dc-2170-4d8b-b774-e154d31c4be1" name="queryManager"/>
  </source-objects>

```

```

</source-objects>
<target-objects>
  <obj type="SUCCESS" valId="971b35cb-0e07-4cd2-808d-24372e3fc5d2" name=""/>
</target-objects>
</trace>

```

G.27 AssociationEndsToPortName

```

<trace>
  <mapping-applied>AssociationEndsToPortName</mapping-applied>
  <source-objects>
    <obj type="AssociationEnd" valId="5f9286d8-2a57-42c0-ae3b-355148d18c6e" name="accountManager"/>
    <obj type="AssociationEnd" valId="0efc70db-a34d-4957-959b-f17720ceb291" name="logger"/>
  </source-objects>
  <target-objects>
    <obj type="SUCCESS" valId="e6031063-a83b-412c-b145-2496d1b6a07c" name=""/>
  </target-objects>
</trace>

```

G.28 AssociationEndToAssociationEnd

```

<trace>
  <mapping-applied>AssociationEndToAssociationEnd</mapping-applied>
  <source-objects>
    <obj type="AssociationEnd" valId="7d446456-015a-43db-a36e-bd2f25e15217" name="manager"/>
  </source-objects>
  <target-objects>
    <obj type="AssociationEnd" valId="81602fd7-7a65-4573-aade-4267fc3a3c4e" name="manager"/>
  </target-objects>
</trace>

```