A Rule-based Approach for Software Traceability on Product Family Systems

Waraporn Jirapanthong

Submitted for the degree of Doctor of Philosophy

City University
Department of Computing

October 2006

แด่ พ่อ และ แม่ ผู้คอยสอนให้รู้และให้โอกาสที่จะเรียน

เพื่อ น้องสาว และ ครอบครัวที่รัก ผู้คอยให้ความเชื่อมั่น

Contents

ACKNOWLEDGEMENT ..... **XV**

DECLARATION ..... **XVII**

ABSTRACT ..... **XIX**

CHAPTER 1

INTRODUCTION **1**

1.1. Hypothesis 6
1.2. Problem Definition and Objectives 7
1.3. Contribution of the Thesis 8
1.4. Thesis Outline 9

PART I: LITERATURE REVIEW **11**

CHAPTER 2

SOFTWARE TRACEABILITY **13**

2.1. Definition of Traceability 13
2.2. Benefits with Software Traceability 15
2.2.1. Economic Aspects 15
2.2.2. Different Use of Software Traceability 17
2.3. Problems with Software Traceability 25
2.4. Reference Models and Classification for Traceability Relations 27
2.4.1. Reference Models 27
2.4.2. Classification of Traceability Relations 32
2.5. Approaches for Establishing Traceability Relations 45
2.5.1. Manual Establishment of Traceability Relations 45
2.5.2. Semi-Automatic Establishment of Traceability Relations 48
2.5.3. Fully Automatic Establishment of Traceability Relations 51
2.6. Representation, Recording, and Maintenance of Traceability Relations 54
2.6.1. Identifier Technique 54
2.6.2. Tagging Technique 55
2.6.3. Indexing Technique 55
2.6.4. Table Technique 56
2.6.5. Mapping Graph Technique 57
2.6.6. Mark-up Technique 57
2.6.7. Hyperlink Technique 58
2.7. Traceability Commercial Tools 60
2.7.1. General-Purpose Tools 60
2.7.2. Specific-Purpose Tools of Requirements Management 61
2.8. Summary 63

CHAPTER 3

PRODUCT FAMILY SYSTEMS 65

3.1. Introduction to Product Family 65
3.1.1. Terminologies in Product Family 66
3.2. Problems of the Establishment and Maintenance of Product Family Systems 68
3.3. Activities in the Process of Product Family System Development 71
3.3.1. Domain Engineering 73
3.3.2. Application Engineering 78
3.4. Methodologies for the Development of Product Family Systems 80
3.4.1. Object-Oriented Methodologies 80
3.4.2. Feature-Oriented Methodologies 89
3.5. Techniques for the Development of Product Family Systems 93
3.5.1. Use Cases 93
3.5.2. UML Modeling 94
3.5.3. Feature Modeling 95
3.5.4. Architecture Description and Component-based Languages 97
3.6. Supporting Tools for Product Family Systems 99
3.7. Traceability of Product Family Systems 105
3.7.1. Existing Approaches for Traceability Generation in Product Family Systems 105
3.7.2. Issues of Traceability Activities in Product Family Systems 108
3.8. Summary 108

PART II: THE APPROACH 109

CHAPTER 4

TRACEABILITY REFERENCE MODEL 111

4.1. Introduction 111
4.2. Product Family Software Artefacts 112
4.3. Traceability Relations 133
4.4. Summary 148

CHAPTER 5

TRACEABILITY FRAMEWORK 149

5.1. Overview of The Traceability Generation Process 149
5.2. Traceability Rules and Relations 155
5.3. Extended Functions 174
5.3.1. Functions in XQuery 174
5.3.2. Functions In Java 182
5.4. Summary 186

CHAPTER 6

XTRAQUE TOOL 187

6.1. Overview 187
6.2. User Interfaces 192
6.2.1. Specifying the Scope of Traceability Generation 193
6.2.2. Specifying Types of Documents and Relationships 194
6.2.3. Specifying Particular Documents and Relationship Types ... 199
6.2.4. Editing and Testing XQuery Statements 202
6.3. Summary 204

CHAPTER 7

MOBILE PHONE SYSTEMS – CASE STUDY 205

7.1. Overview of the Case Study 205
7.2. Documents in the Mobile-Phone Systems 213
7.2.1. Feature Model of Mobile-Phone Systems 213
7.2.2. Subsystem Model of Mobile-Phone Systems 213
7.2.3. Process Models of Mobile-Phone Systems 215
7.2.4. Module Models of Mobile-Phone Systems 217
7.2.5. Use Cases, Class, Statechart, and Sequence Diagrams of Mobile-Phone Members 223
7.3. Summary 223

PART III: EVALUATION AND CONCLUSION 225

CHAPTER 8

EVALUATION AND ANALYSIS 227

8.1. Evaluation Overview 227
8.1.1. Scenario 1: The creation of a new product member from existing product family 230
8.1.2. Scenario 2: The creation of product family from already existing products 233
8.1.3. Scenario 3: Changes to a product member in a product family 236
8.1.4. Scenario 4: Changes to the core assets of a product family 237
8.1.5. Scenario 5: Impact of changes to the core assets of a product family and product members 239
8.2. Evaluation Results and Analysis 240
8.3. Summary 249

CHAPTER 9

CONCLUSIONS AND FUTURE WORK 251

9.1. Overall Conclusions 251
9.2. The Findings 253
9.3. Future Work 256
9.4. Final Remarks 257

APPENDICES 259

APPENDIX A – XML SCHEMAS 261

A.1. XML SCHEMA FOR DIRECT TRACEABILITY RULES 262
A.2. XML SCHEMA FOR INDIRECT TRACEABILITY RULES 264
A.3. XML SCHEMA FOR FEATURE MODEL 266
A.4. XML SCHEMA FOR USE CASE 271
A.5. XML SCHEMA FOR SUBSYSTEM MODEL 274
A.6. XML SCHEMA FOR PROCESS MODEL 276
A.7. XML SCHEMA FOR MODULE MODEL 279

APPENDIX B – TRACEABILITY RULES 281

B.1. DIRECT TRACEABILITY RULES 281
B.2. INDIRECT TRACEABILITY RULES 316

APPENDIX C – EXTENDED XQUERY FUNCTIONS 322

C.1. GETTRANSITIONINSTATE 322
C.2. GETSTATEINSTATE 322
C.3. GETMESSAGEINSEQ 322
C.4. GETOBJECTINSEQ 323
C.5. GETCLASSID 323
C.6. GETCLASSOBJECTINSEQ 323
C.7. GETPARENTFEATURE 323
C.8. GETCHILDRENFEATURE 324
C.9. GETFEATUREOFSUBSYSTEM 324
C.10. GETOPERATIONINSEQ 324
C.11. GETOPERATIONINMODEL 324
C.12. GETSTATEOFOPERATIONINSTATE 325
C.13. GETCLASSINCLASS 325
C.14. GETPARENTOFVARIANTCLASSES 326
C.15. GETPARENTOFVARIANTFEATURES 327
C.16. GETPARENTCLASS 328

APPENDIX D – EXAMPLE DOCUMENTS IN MOBILE-PHONE SYSTEMS 329

D.1. USE CASE – PM1 330
D.2. CLASS DIAGRAM – PM1 341
D.3. SEQUENCE DIAGRAM – PM1 342
D.4. STATECHART DIAGRAM – PM1 346

BIBLIOGRAPHY 347

List of Figures

FIGURE 2- 1: REPRESENTING TRACEABILITY BY USING IDENTIFIERS.....55
FIGURE 2- 2: REPRESENTING TRACEABILITY BY TAGGING ATTRIBUTES.....55
FIGURE 2- 3: REPRESENTING TRACEABILITY BY INDEXING.....56
FIGURE 2- 4: REPRESENTING TRACEABILITY BY TABLE.....56
FIGURE 3- 1: ACTIVITIES IN SOFTWARE PRODUCT LINE ENGINEERING (NORTHROP 2002).....73
FIGURE 3- 2: DIFFERENT NOTATIONS FOR DIFFERENT TYPES OF A FEATURE: (A) (KANG ET AL. 1990); (B) (GRISS ET AL. 1998, KANG ET AL. 1998); AND (C) (SVAHNBERG ET AL. 2001).....96
FIGURE 4- 1: THE FEATURE MODEL OF THE MOBILE PHONE 116
FIGURE 4- 2: FEATURES IN TEXTUAL SPECIFICATION LANGUAGE (KANG ET AL. 1998). 117
FIGURE 4- 3: FEATURE MODEL FOR MOBILE PHONE SYSTEMS..... 118
FIGURE 4- 4: USE CASE *SENDING A MESSAGE*..... 120
FIGURE 4- 5: SUBSYSTEM MODEL OF MOBILE PHONE SYSTEMS 122
FIGURE 4- 6: EXAMPLE OF SUBSYSTEM MODEL 123
FIGURE 4- 7: *SMS* PROCESS MODEL FOR *MESSAGING* SUBSYSTEM..... 125
FIGURE 4- 8: SMS PROCESS MODEL FOR *MESSAGING* SUBSYSTEM..... 127
FIGURE 4- 9: THE MODULE MODEL FOR *SHORT MESSAGING SERVICE (SMS) CONTROL* PROCESS 129
FIGURE 4- 10: MODULE MODEL FOR *SHORT MESSAGING SERVICE SMS CONTROL* PROCESS..... 130
FIGURE 4- 11: AN EXTRACT OF A CLASS DIAGRAM FOR PRODUCT MEMBER *PM1* ... 131
FIGURE 4- 12: A STATECHART DIAGRAM FOR A *DIGITAL_CAMERA* CLASS 132
FIGURE 4- 13: AN EXTRACT OF A SEQUENCE DIAGRAM OF *TAKING A PHOTO*..... 132
FIGURE 4- 14: EXAMPLES OF *SATISFIABILITY, DEPENDENCY, REFINEMENT, CONTAINMENT,* AND *SIMILAR* TRACEABILITY RELATIONS 144
FIGURE 4- 15: EXAMPLES OF *IMPLEMENTS, OVERLAPS, EVOLUTION,* AND *DIFFERENT* TRACEABILITY RELATIONS..... 145
FIGURE 5- 1: OVERVIEW OF TRACEABILITY GENERATION PROCESS..... 151
FIGURE 5- 2: *TRACEABILITY GENERATOR* 152
FIGURE 5- 3: TRACEABILTIY RULE TEMPLATE..... 156
FIGURE 5- 4: EXAMPLE OF *CONTAINMENT* TRACEABILITY RULE 160
FIGURE 5- 5: EXAMPLE OF *SIMILAR* TRACEABILITY RULE..... 160
FIGURE 5- 6: RESULT OF *CONTAINMENT* TRACEABILITY RELATIONS 161
FIGURE 5- 7: EXAMPLE OF USE CASE UC 2 *TRANSMITTING MESSAGE*..... 162
FIGURE 5- 8: RESULT OF SIMILAR TRACEABILITY RELATION..... 162FIGURE 5- 9: EXAMPLES OF TRACEABILITY RELATIONS (REPETITIVE TO FIGURE 4-14) 164
FIGURE 5- 10: EXAMPLES OF TRACEABILITY RELATIONS (REPETITIVE TO FIGURE 4-15) 165
FIGURE 5- 11: EXAMPLE OF *DEPENDENCY* TRACEABILITY RULE 166
FIGURE 5- 12: EXAMPLE OF *REFINEMENT* TRACEABILITY RULE 167

FIGURE 5- 13: EXAMPLE OF *SATISFIABILITY* TRACEABILITY RULE 168
FIGURE 5- 14: EXAMPLE OF *IMPLEMENTS* TRACEABILITY RULE 169
FIGURE 5- 15: EXAMPLE OF *DIFFERNT* TRACEABILITY RULE 170
FIGURE 5- 16: EXAMPLE OF *OVERLAPS* TRACEABILITY RULE 172
FIGURE 5- 17: EXAMPLE OF *OVERLAPS* TRACEABILITY RULE 173
FIGURE 5- 18: A STRUCTURE OF AN USER-DEFINED FUNCTION 175
FIGURE 5- 19: GETTRANSITIONINSTATE FUNCTION 175
FIGURE 5- 20: EXTRACT OF A STATECHART DIAGRAM 176
FIGURE 5- 21: GETSTATEINSTATE FUNCTION 176
FIGURE 5- 22: GETTRANSITIONINSTATE FUNCTION 176
FIGURE 5- 23: EXTRACT OF A SEQUENCE DIAGRAM 177
FIGURE 5- 24: GETOBJECTINSEQ FUNCTION 177
FIGURE 5- 25: GETCLASSOBJECTINSEQ FUNCTION 177
FIGURE 5- 26: GETCLASSINCLASS FUNCTION 178
FIGURE 5- 27: EXTRACT OF A CLASS DIAGRAM 178
FIGURE 5- 28: GETPARENTFEATURE FUNCTION 178
FIGURE 5- 29: EXTRACT OF FEATURE MODEL 179
FIGURE 5- 30: GETCHILDRENFEATURE FUNCTION 179
FIGURE 5- 31: GETFEATUREOFSUBSYSTEM FUNCTION 179
FIGURE 5- 32: GETOPERATIONINSEQ FUNCTION 180
FIGURE 5- 33: GETOPERATIONINCLASS FUNCTION 180
FIGURE 5- 34: GETSTATEOFOPERATIONINSTATE FUNCTION 180
FIGURE 5- 35: GETPARENTOFVARIANTFEATURES FUNCTION 181
FIGURE 5- 36: GETPARENTOFVARIANTCLASSES FUNCTION 181
FIGURE 5- 37: GETPARENTCLASS FUNCTION 182
FIGURE 5- 38: GETCLASSID FUNCTION 182
FIGURE 5- 39: THE DECLARATION OF A NAMESPACE REFERRING TO EXTRA FUNCTIONS IN JAVA PACKAGE 183
FIGURE 5- 40: CALLING AN EXTENDED FUNCTION IMPLEMENTED IN JAVA 183
FIGURE 6- 1: THE ARCHITECTURE OF XTRAQUE TOOL 189
FIGURE 6- 2: AN XTRAQUE INTERFACE FOR SPECIFYING A SCOPE OF TRACEABILITY GENERATION 193
FIGURE 6- 3: EXAMPLE INTERFACE DEMONSTRATING SPECIFYING THE SCOPE OF TRACEABILITY GENERATION BETWEEN DOCUMENTS AT THE LEVELS OF PRODUCT LINE AND TWO PRODUCT MEMBERS, *MODEL PM1* AND *MODEL PM2* 194
FIGURE 6- 4: AN XTRAQUE INTERFACE FOR SPECIFYING TYPES OF DOCUMENT ARTIFACTS AND RELATIONSHIPS ACCORDING TO TRACING BETWEEN THE PRODUCT-LINE AND TWO PRODUCT MEMBERS 196
FIGURE 6- 5: AN XTRAQUE INTERFACE FOR SPECIFYING TYPES OF DOCUMENT ARTIFACTS AND RELATIONSHIPS ACCORDING TO TRACING BETWEEN TWO PRODUCT MEMBERS, *MODEL PM1* AND *MODEL PM2* 198
FIGURE 6- 6: EXAMPLE INTERFACE DEMONSTRATING SPECIFYING OF TYPES OF DOCUMENT ARTIFACTS AND RELATIONSHIPS 199
FIGURE 6- 7: AN XTRAQUE INTERFACE FOR SPECIFYING PARTICULAR DOCUMENTS AND RELATIONSHIPS ACCORDING TO THE SPECIFIED CRITERIA FROM THE PREVIOUS INTERFACE (FIGURE 6-4) 200

FIGURE 6- 8: AN XTRAQUE INTERFACE FOR SPECIFYING PARTICULAR DOCUMENTS AND RELATIONSHIPS ACCORDING TO THE SPECIFIED CRITERIA FROM THE PREVIOUS INTERFACE (FIGURE 6-5) 201
FIGURE 6- 9: EXAMPLE INTERFACE DEMONSTRATING: DISPLAYING THE CONTEXT OF AN XML-BASED DOCUMENT; AND SELECTION OF DOCUMENTS TYPES AND RELATIONSHIP TYPES TO BE TRACED 202
FIGURE 6- 10: AN XTRAQUE INTERFACE FOR CREATING AND VERIFYING THE TRACEABILITY RULES 203
FIGURE 6- 11: EXAMPLE INTERFACE FOR CREATING AND VERIFYING TRACEABILITY RULES 204
FIGURE 7- 1: *INTERNET APPLICATION* PROCESS MODEL 217
FIGURE 7- 2: MODULE MODEL FOR *INTERNET APPLICATION* PROCESS MODEL 222
FIGURE 8- 1: SCENARIO 1 231
FIGURE 8- 2: SCENARIO 2 233
FIGURE 8- 3: SCENARIO 3 236
FIGURE 8- 4: SCENARIO 4 238
FIGURE 8- 5: SCENARIO 5 239
FIGURE 8- 6: TRACEABILITY RELATIONS DETECTED BY THE TRACEABILITY USER AND XTRAQUE (A), BY TRACEABILITY USER (B), AND BY XTRAQUE (C) IN FIVE EXPERIMENTS 245
FIGURE 8- 7: (A) PRECISION AND RECALL FIGURES OF EACH SCENARIO; (B) COMPARISON OF PRECISION AND RECALL FIGURES FROM FIVE SCENARIOS 247
FIGURE D- 1: USE CASE *SENDING A MESSAGE* 332
FIGURE D- 3: USE CASE *TAKING A PICTURE* 337
FIGURE D- 6: A SEQUENCE DIAGRAM *MAKING A CALL* 342
FIGURE D- 7: A SEQUENCE DIAGRAM *SENDING DATA* 343
FIGURE D- 8: A SEQUENCE DIAGRAM *TAKING A PHOTO* 344
FIGURE D- 9: A SEQUENCE DIAGRAM *TRANSFERING DATA* 345
FIGURE D- 10: A STATECHART DIAGRAM OF PRODUCT MEMBER PM1 346

List of Tables

TABLE 2- 1: DIFFERENT TRACEABILITY RELATIONSHIPS BETWEEN DIFFERENT ARTEFACTS 41
TABLE 2- 2: COMPARISON OF TECHNIQUES FOR TRACEABILITY REPRESENTATION 59
TABLE 3- 1 PRESENTS THE CLASSIFICATION OF RELATIONSHIPS BETWEEN FEATURES: 97
TABLE 3- 2: COMPARISON APPROACHES FOR PRODUCT FAMILY SYSTEM DEVELOPMENT 101
TABLE 4- 1: DOCUMENTS USED IN OUR APPROACH 114
TABLE 4- 2: SUMMARY OF TRACEABILITY RELATION GROUPS 134
TABLE 4- 3: TRACEABILITY REFERENCE MODEL 147
TABLE 5- 1: VARIATION OF *CONTAINSINDISTANCE* FUNCTION WITH DIFFERENT PARAMETERS 184
TABLE 5- 2: A LAYOUT OF *FINDSYNONYM* FUNCTION 185
TABLE 5- 3: A LAYOUT OF *STRINGNOSPACE* FUNCTION 185
TABLE 5- 4: A LAYOUT OF *SETOF* FUNCTION 186
TABLE 5- 5: A LAYOUT OF *CHECKDISTANCECONTROL* FUNCTION 186
TABLE 6- 1: ICONS IN PANEL (A) 197
TABLE 7- 1: FUNCTIONALITIES OF MOBILE PHONE MEMBERS 208
TABLE 7- 2: SPECIFICATIONS OF MOBILE PHONE MEMBERS 209
TABLE 7- 3: MODULES FOR SHORT MESSAGING SERVICE (SMS) CONTROL PROCESS MODEL 218
TABLE 7- 4: MODULES FOR INTERNET APPLICATION PROCESS MODEL 220
TABLE 8- 1: DOCUMENTS AND TRACEABILITY RELATIONS FOR SCENARIO 1 232
TABLE 8- 2: DOCUMENTS AND TRACEABILITY RELATIONS FOR SCENARIO 2 235
TABLE 8- 3: DOCUMENTS AND TRACEABILITY RELATIONS FOR SCENARIO 3 237
TABLE 8- 4: DOCUMENTS AND TRACEABILITY RELATIONS FOR SCENARIO 4 238
TABLE 8- 5: DOCUMENTS AND TRACEABILITY RELATIONS FOR SCENARIO 5 240
TABLE 8- 6: SUMMARY OF DOCUMENTS, FILES, TRACEABILITY RULE TEMPLATES, AND INSTANTIATED TRACEABILITY RULES USED IN THE EXPERIMENTS 241
TABLE 8- 7: SUMMARY OF TRACEABILITY RELATIONS DETECTED IN SCENARIO 1 242
TABLE 8- 8: SUMMARY OF TRACEABILITY RELATIONS DETECTED IN SCENARIO 2 242
TABLE 8- 9: SUMMARY OF TRACEABILITY RELATIONS DETECTED IN SCENARIO 3 243
TABLE 8- 10: SUMMARY OF TRACEABILITY RELATIONS DETECTED IN SCENARIO 4 243
TABLE 8- 11: SUMMARY OF TRACEABILITY RELATIONS DETECTED IN SCENARIO 5 243
TABLE 8- 12: SUMMARY OF TRACEABILITY RELATIONS DETECTED IN THE EXPERIMENTS 244
TABLE 8- 13: PRECISION AND RECALL RATES (%) 245
TABLE 8- 14: SUMMARY OF RECALL AND PRECISION RATES ACHIEVED BY SEVERAL EXISTING TRACEABILITY APPROACHES 248

Acknowledgement

I would like to express my sincere gratitude to my supervisor Dr. Andrea Zisman for her insight and support throughout all stages of thesis. I would also like to thank my reviewers, Dr. Bill Karakostas and Prof. Dr. Mike Mannion for their constructive comments. I am very grateful to Dhurakijpundij University, Thailand, for the financial support they provided during the course of this study.

I am grateful to Neek Fenwick and Dr. Kelly Androutsopoulos to read drafts of the thesis and providing valuable comments. I would also like to thank Dr. Penny Noy and Dr. Edward Parkinson for their general advice, and Dr. Thanwadee Sunetnanta for her constructive advice and feedback.

I would like to thank people at City U., especially Gilberto Cysneiros for his invaluable friendship and encouragement, Cristina Arciniegas for her lovely support, and Khaled Mahbub for his helpful feedback. Many thanks are due to Annie Benavidis for his loving support. Thanks also go to Mark Firman, Tshiamo Motshegwa, and Greek family for providing an appealing environment for working.

Finally, I would like to thank my parents for their untiring love, support, and understanding through the years, my sister for encouraging me with her belief in me, my brothers for their wonderful backup. Special thanks are given to Rathpol Bumrungkittikul for a wonderful support through the toughest of times and for always believing in me.

Declaration

Some of the material in this thesis has been previously published in the following papers:

- Jirapanthong, W. 2004. Towards a Traceability Approach for Product Family Systems. *International Software Product Lines Young Researchers Workshop in International Software Product Line Conference*, Boston, MA.

- Jirapanthong, W., and A. Zisman. 2004. Traceability for Product Family Systems: An XQuery Approach. *International Workshop on Requirements Reuse in System Family Engineering in International Conference on Software Reuse*, Madrid, Spain.

- Jirapanthong, W., and A. Zisman. 2005. Supporting Product Line Development through Traceability. *12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, Taipei, Taiwan.

- Jirapanthong, W., and A. Zisman. 2006. XTraQue: Traceability for Product Line Systems. *Software and Systems Modeling* (under review).

Abstract

Software traceability has been recognized as an important activity in software system development. Traceability relations can improve the quality of a system being developed, as well as reduce the time and cost associated with the development. In particular, traceability relations can facilitate the development process, reuse of parts of the systems by comparing artefacts, validation that a system meets its requirements, understanding the rationale for certain design and implementation decisions, and analysis of the implications of changes in the system. However, support for traceability in software engineering environments and tools are not always adequate. In addition, automatic generation and maintenance of traceability relations are not easy tasks.

In contrast, product family systems, in which software systems share a common set of features and new product members can be built around a set of reusable artefacts, is considered an important paradigm for software system engineering. Despite its importance and advances in the area, the support for common and variable aspects among applications and the engineering of reusable and adaptable components are difficult tasks. This is mainly due to the large number and heterogeneity of documents generated during the development of product family systems.

The underlying principle of this thesis is to use of traceability to support the difficulties associated with product family systems. More specifically, traceability can assist with the identification of common and variable functionalities of the product members, reduction of inconsistencies between product members, reuse of available core assets, and establishment of relationships between product members and product family architectures.

The thesis presents a *traceability reference model* with nine different types of traceability relations for eight different types of documents generated in feature-based object-oriented methodologies, and a *rule-based approach* to allow automatic generation of traceability relations in documents produced during the development of software product family systems. The documents are represented in XML and the different types of traceability relations are identified by using traceability rules expressed in an extension of XQuery. The textual sentences of the XML documents are annotated with part-of-speech assignments indicating the grammatical roles of the various words in the sentence. The traceability rules are based on (i) the semantic of the documents being compared, (ii) the various types of traceability relations in the product family domain, (iii) the grammatical roles of the words in the textual parts of the documents, and (iv) synonyms and distance of words being compared in a text.

A prototype tool called *XTraQue* has been developed to demonstrate and evaluate automatic generation of traceability relations. We use a case study from mobile phone domain to illustrate the feasibility and applicability of the approach and to evaluate the work in terms of recall and precision measures.

Chapter 1

Introduction

In recent years we have been experiencing the proliferation of a large number of software systems that share a common set of features and have also their own distinct characteristics. Examples of such systems are found in the telecommunication domain in which products including personal digital assistants (PDAs), mobile phones, and pagers have many common characteristics. Other examples are found in the automotive, electronics, medical imaging, and elevator control domains. These systems are known in the literature as *product family systems* (Ardis and Weiss 1997, Bass et al. 2003, CAFE 2003, Clements and Northrop 2002, Clements and Northrop 2004, Staudenmayer and Perry 1996, Weiss and Lai 1999) and are characterized as being software systems that share a common set of features and are developed based on the reuse of core assets and addition of new functionalities.

Various methodologies and approaches have been proposed to support the development of software systems based on product family system development. Examples of these methodologies and approaches are FODA (Kang et al. 1990), FeatuRSEB (Griss et al. 1998), CAFÉ (CAFE 2003), FAST (Weiss 1995), FORM (Kang et al. 1998), PuLSE (Bayer et al. 1999), and KobrA (Atkinson et al. 2000).

The above methodologies and approaches are also known as *domain-engineering* approaches and emphasise a group of related applications in a domain, instead of single applications (Northrop 2002). Their main focus is the identification and analysis of commonality and variability principles among applications in a domain in order to engineer reusable and adaptable components and, therefore, support

product family system development. However, despite its importance and advances in the area, the support for common and variable aspects among applications and the engineering of reusable and adaptable components are not easy tasks. This is mainly due to the large number and heterogeneity of documents generated during the development of product family systems. Other difficulties are concerned with the (a) necessity of having a basic understanding of the variability consequences during the different development phases of software products by all involved parties (Sinnema 2004, Svahnberg and Bosch 2000, Thiel and Hein 2002), (b) necessity of establishing relationships between product members and product family artefacts, and relationships between product members artefacts (Bayer and Widen 2002, Mohan and Ramesh 2002), (c) poor support for capturing, designing, and representing requirements for the whole product family and for specific product members (Fantechi et al. 2004, Mannion et al. 2000), (d) poor support for handling complex relations among product members (Bayer and Widen 2002, Mohan and Ramesh 2002), and (e) poor support for maintaining information about the development process (Meyer 1998).

In this thesis, we advocate the use of software traceability to support the difficulties associated with product family system development. Software traceability has been recognized as an important activity in software system development in which traceability relations are generated between software artefacts and between stakeholders and software artefacts (Gotel and Finkelstein 1995). Software traceability can improve the quality of the product being developed and reduce cost and time of development (Gotel and Finkelstein 1994, Pohl 1996a, Ramesh and Jarke 2001). It supports software developers in many activities such as verifying requirements, ensuring completeness and supporting evolution of software systems, enhancing maintainability, and maintaining consistency of software systems, understanding the rationale for certain design and implementation decisions, and analysis of the implications of changes in the system.

The need for traceability is due to the large amount and heterogeneity of software artefacts that are generated during the development of software systems and the

lack of formalism when developing software systems. As affirmed by (Finkelstein 1991), if formal methodologies are used during the development of software system and systems are generated from formal specifications without requiring changes to the code, then there is no need for traceability.

Traceability can support the difficulties associated with product family systems since traceability relations can assist with the (i) identification of common and variable functionalities in the product members, (ii) reduction of inconsistencies between product members, (iii) reuse of core assets that are available in the product family system, (iv) maintenance of historical information of the development process, and (v) establishment of relationships between core assets of product family systems and product members specification documents.

Although many approaches for software traceability have been proposed, support for traceability in software engineering environments and tools are not always adequate (Ramesh 2001). For example, (Bayer 2001; RTM; DOORS) assume that traceability relations should be established manually which are error-prone, difficult, time consuming, expensive, complex, and limited in expressiveness. Attempts have been made to alleviate the issues associated with manual techniques and, more recently, other approaches have been proposed to support semi-automatic or fully automatic generation of traceability relations (Antoniol et al. 2002, Cleland-Huang et al. 2002b, Egyed and Grunbacher 2002, Marcus and Meletic 2003, Pinheiro 2000, Pohl 1996a, Ramesh and Dhar 1992, Sherba et al. 2003b, Spanoudakis et al. 2004).

However, in the majority of those approaches, the generated traceability relations do not have enough semantic meanings to support the full benefits that are provided by software traceability. Exceptions are found in the approaches proposed by (Alexander 2003, Gotel and Finkelstein 1994, Knethen et al. 2002, Pohl 1996b, Ramesh and Jarke 2001, Sherba et al. 2003a, Spanoudakis et al. 2004) that present different types of traceability relations. However, these approaches do not cover product family systems and do not tackle the similar and different perspectives between software artefacts. Moreover, traceability practice becomes more difficult

and ambiguous in product family systems due to their rigidness and complexity (Bayer and Widen 2002, CAFE 2003, Lago et al. 2004). Although the use of traceability relations to support product family system development has been advocated in (Atkinson et al. 2002, Bayer and Widen 2002, Berg and Bishop 2005, CAFE 2003, Coriat et al. 2000, Dick 1999, ESAPS, Jazayeri et al. 2000, Kim et al. 2005, Lago et al. 2004, Mohan and Ramesh 2002, Plankl and Bockle 2001, PLP, Riebisch and Philippow 2001), the majority of these approaches focus on traceability meta models and do not provide ways of generating traceability relations automatically.

In this thesis, we present a rule-based approach to allow automatic generation of traceability relations between documents created during the development of product family systems. Our work largely extends the work from (Spanoudakis et al. 2004, Zisman et al. 2002a). This previous work proposes a rule-based approach to allow an automatic generation of traceability relations between different types of requirements documents such as customer requirements specifications, use case specification, and analysis object models. In this work, three different types of traceability relations have been proposed, namely *overlaps*, *requires*, and *realizes* relations.

In contrast, the work in this thesis (a) describes a traceability reference model with nine different types of traceability relations for eight different types of feature-based object-oriented documents, and (b) supports automatic generation of all these nine traceability relations between software artefacts of product family's core assets (i.e. requirements and software architecture) and software artefacts of product members.

The documents used in our approach are based on feature-based object-oriented methodologies. A feature-based approach supports domain analysis and design, while an object-oriented approach assists with the development of various product members. Our approach applies the *feature oriented reuse method* (FORM) (Kang et al. 1998) and the unified modeling language (UML) (UML). The use of FORM is due to several reasons, namely (a) practicality – FORM has been applied to several

industrial product family systems such as elevator control systems, electronic bulletin board systems, yard automation systems, and PBX; (b) maturity – FORM is an extension of *feature-oriented approach to domain analysis* (FODA) (Kang et al. 1990) and includes domain design and implementation phases; (c) extensibility – FORM is extensible and can be extended to accommodate the object-oriented techniques for reusable components in the architecture of product family; (d) simplicity – FORM is based on a feature modeling which becomes a common technique in software engineering process and has provided a tool called ADASAL (ASADAL) to assist developers during domain analysis and design.

On the other hand, UML has been chosen as the object-oriented methodology due to its (a) maturity – many approaches and methodologies have been applied with UML over the years; (b) compliance – UML is the de facto modeling language for software analysis and design in object-oriented systems; and (c) practicality – many commercial software tools support UML modeling.

The documents used in our approach include *feature*, *subsystem*, *process*, and *module* models to specify the information of core assets, and *use cases*, *class*, *statechart*, and *sequence* diagrams to specify the information of product members. In our approach, the documents are represented in XML and the different types of traceability relations are identified by using traceability rules expressed in an extension of XQuery (XQuery). The textual sentences of the XML documents are annotated with part-of-speech assignments indicating the grammatical roles of the various words in the sentence. These grammatical roles are used to assist with the matching of textual terms in the documents. The traceability rules are classified as *direct rules*, i.e. rules that support the creation of traceability relations that do not depend on the existence of other relations; and *indirect rules*, i.e. rules that require the existence of previously generated relations. In both types of rules, when a matching expected by a rule is found, a traceability relation of the type specified in the rule is created between parts of the documents being compared by the rule.

A prototype tool called *XTraQue* has been implemented in order to illustrate and evaluate the work. The evaluation of the work has been performed in a mobile phone case study. The results of this evaluation are encouraging and better than other approaches that support automatic generation of traceability relations.

The remainder of this chapter describes the hypothesis, problem definition and objectives, contribution, and thesis outline.

1.1. Hypothesis

The hypothesis of the work presented in this thesis is that:

It is possible to automatically generate traceability relations for product family systems.

We advocate the fact that traceability relations should be generated automatically, since manual traceability generation is error-prone, time consuming, and costly, leading to the situation in which traceability is rarely established (Bayer and Widen 2002, CAFE 2003, Lago et al. 2004). This is also the case for product family systems in which large numbers of different artefacts are created during the development process. We expect to reduce the effort of activity domain and application engineering that require traceability practice during the development of product family systems by enabling automatic support for traceability generation process. Examples of the activities in domain and application engineering which requires using traceability relations are such as (i) verifying requirements; (ii) understanding common and variable aspects between product members; (iii) understanding the rationale for certain requirement and design decisions, and analysis of the implications of changes in product family systems; (iv) ensuring completeness and supporting evolution of product family systems; and (v) maintaining consistency of product members in a family.

1.2. Problem Definition and Objectives

More specifically, the work presented in this thesis is aimed to tackle two main problems in the areas of software traceability and product family systems as discussed below:

I. The Lack of Automatic Support for Traceability between Artefacts of Product Family Systems

As discussed, there is poor support or mechanisms to establish traceability in product family systems (Fantechi et al. 2004, Lago et al. 2004). Although there are many approaches to support establishment of traceability in single product software systems, these approaches cannot be used to support traceability of product family systems (Antoniol et al. 2002, Clements and Northrop 2002, Egyed and Grunbacher 2002, Gotel and Finkelstein 1994, Marcus and Meletic 2003, Pohl 1996b, Ramesh and Jarke 2001, Sherba et al. 2003a). On the other hand, although some recent approaches have been proposed to support the development of product family systems and tackle the traceability issue, they do not provide automatic generation of traceability relations (Bayer et al. 1999, CAFE 2003, Coriat et al. 2000, ESAPS, Hull et al. 2002, Kim et al. 2005, Plankl and Bockle 2001, Riebisch and Philippow 2001).

Our work is focused on establishing traceability relations automatically for artefacts generated during domain analysis and design of product family systems.

II. The Difficulty of Identifying the Semantics of Traceability Relations in the Domain of Product Family Systems

Due to a large size, diversity, and complexity of the artefacts in product family systems, it is difficult to identify the semantics of traceability relations in these artefacts (Bayer and Widen 2002, Mohan and Ramesh 2002). This is particularly true in the case of common and variable relations between product family artefacts that are required to be considered during the development of product family systems

(Bayer and Widen 2002, Coriat et al. 2000, Kang et al. 1998, Mohan and Ramesh 2002, Riebisch and Philippow 2001).

Our work is aimed to generate traceability relations that provide semantics to the relationships. Some of traceability relations are concerned with common and variable aspects between artefacts in product family systems, while other relations are concerned with satisfaction, containment, dependency, evolution, overlap, implementation, and refinement aspects.

1.3. Contribution of the Thesis

The main contributions of our work are:

I. Traceability Reference Model for Product Family Systems

We have investigated which artefacts are playing the main roles in the process of product family system development and classified relationships which exist between those artefacts. The concepts and motivation of the classification of traceability relations in the domain of product family systems have been initially proposed in (Jirapanthong 2004, Jirapanthong and Zisman 2004). The traceability reference model has been initially described in (Jirapanthong and Zisman 2005) and also appeared in (Jirapanthong and Zisman 2006).

II. Rule-Based Approach for Generating Traceability Relations

We apply a rule-based approach for automatically establishing traceability relations according to the traceability reference model. The concept of the rule-based approach for traceability generation has been initially presented in (Jirapanthong 2004, Jirapanthong and Zisman 2004) and also appeared in (Jirapanthong and Zisman 2005, Jirapanthong and Zisman 2006). The rule-based approach takes into consideration:

(a) the semantics of document types;

(b) the types of traceability relations;

(c) the part-of-speech of the words in textual sentences in the documents; and

(d) the synonym and distance of words in textual sentences in the documents.

III. Evaluation of the Rule-based Traceability Approach for Product Family Systems

We have demonstrated the rule-based approach for generating the traceability relations in the domain of product family systems through five different scenarios. Each scenario presents the experiment of generating the traceability relations that occurs during the process of product family system development. The experiments of the approach are applied with the *XTraQue* tool. The results of the generation of traceability relations in each scenario are evaluated in order to justify the research in this thesis. We have also illustrated the demonstration and evaluation of our work in (Jirapanthong and Zisman 2006).

1.4. Thesis Outline

The remainder of this thesis is organized in three parts composed of eight chapters and four appendices as described below:

Part I: Literature Review

Chapter 2 provides a survey on software traceability, including existing problems and current approaches to support software traceability.

Chapter 3 presents a survey on product family, including the current methods and techniques for product family system development, as well as review of existing approaches for traceability to product family systems.

Part II: The Approach

Chapter 4 presents the traceability reference model, describes the different types of documents used in our work, and introduces the classification of traceability relationship types for product family systems.

Chapter 5 addresses our traceability platform for product family systems. The chapter elaborates the proposed approach including traceability rules, traceability relations, and extra functions.

Chapter 6 discusses *XTraQue*, a prototype tool to allow automatic generation of traceability relations.

Chapter 7 describes a case study of mobile phone systems used to illustrate and evaluate our work. We present the family of mobile-phone products and its members according to the documents of our concern.

Part III: Evaluation and Conclusion

Chapter 8 contains a description of the experiments that we have developed to demonstrate the work and evaluates the experimental results of our work.

Chapter 9 discusses the conclusions of the thesis and directions for future work.

Appendices:

Appendix A describes XML schemas for traceability rules (direct and indirect traceability rules) and XML schemas for some documents of our concern i.e. feature model, use case, subsystem model, process model, and module model.

Appendix B presents the traceability rules used in our work.

Appendix C presents the extra functions that we have implemented in XQuery language.

Appendix D presents examples of documents created for the case study.

Part I: Literature Review

Chapter 2

Software Traceability

This chapter describes a literature of software traceability including current problems, existing approaches, techniques and tools for traceability activities i.e. traceability generation, representation, recording and maintenance, as well as use of software traceability during software development life-cycle. The definition of, benefits with, and current problems of software traceability are given in Section 2.1, Section 2.2, and Section 2.3, respectively. In Section 2.4, we describe types of traceability relations in relation to the types of software artefacts identified for traceability relations. Section 2.5 illustrates existing approaches for traceability generation. In Section 2.6, we summarise techniques for representing, recording, and maintenance of traceability relations. Section 2.7 describes existing tools which are used to support traceability activities.

2.1. Definition of Traceability

The term *traceability* has been initially used as *requirements traceability* and is concerned with the ability to relate requirements with all the other software artefacts generated during the development of software system (Gotel and Finkelstein 1994). More recently, we have been experiencing the use of the term *software traceability*, defined by (Antoniol et al. 2000, Gotel and Finkelstein 1995, Lindvall and Sandahl. 1996, Pohl 1996b, Ramesh and Jarke 2001, Zisman et al. 2002b) as the ability to relate software artefacts created during the life-cycle of software system development such as retrieval documents, requirement specifications, analysis and design models, source codes, and test cases.

Gotel and Finkelstein (Gotel and Finkelstein 1994) proposed traceability relations to be bidirectional relations in which requirements can be associated with other software artefacts in both forward and backward directions. They classified the categories associated with traceability, namely (a) *pre-requirements specification* (pre-RS) traceability, in which traceability relations associate requirements with source of requirements, and (b) *pos-requirements specification* (pos-RS) traceability, in which traceability relations associate requirements with other different artefacts generated in different phases of development life-cycle.

Similar to (Gotel and Finkelstein 1994), Jarke (Jarke 1998) affirmed that the ability to perform traceability can be accomplished by four different types of relations in forward and backward directions. These relations are (a) *forward from the requirements*, which relate a particular requirement forward to design artefacts, (b) *backward to the requirements*, which relate a particular design artefact backward to requirements, (c) *forward to the requirements*, which relate customer's needs or source of requirements forward to requirements, and (d) *backward from the requirements*, which relate requirements backward to customer's needs of source of requirements.

Lawrence and Bohner (Lawrence-Pfleeger and Bohner 1990) have proposed the concepts of *horizontal* and *vertical* traceability. By vertical traceability, they mean the association between different types of artefacts in different phases of software development life-cycle. By horizontal traceability, they mean the association between same types of artefacts in different granularity levels of software development life-cycle. The concepts are applied by (ESAPS) that use a standard V-model to represent different artefacts from different phases in software development and traceability between those types of artefacts such as: (a) vertical traceability between system requirements and system test, between subsystem requirements and subsystem test, and component requirements and component test; (b) horizontal traceability between system requirements and subsystem requirements, and subsystem requirements and component requirements.

2.2. Benefits with Software Traceability

We describe below some of the benefits associated with software traceability. We divide these benefits into two groups. The first group is concerned with economic aspects of software traceability while the second group is concerned with the different ways of using software traceability in the various activities of the software development life-cycle.

2.2.1. Economic Aspects

Software traceability has been well-known as an important activity in software development life-cycle. The desirability of software traceability is worth considering in terms of its expected benefits relative to its cost. The fact is that the cost spent for establishing traceability relations is considered an extra cost during the development of software systems. However, the risk of not performing software traceability during software system development is significant. The costs for establishing software traceability are mainly concerned with two factors: (i) time and (ii) manpower. According to (Leishman and Cook 2002), possible risks of not performing software traceability are: (a) a software system is not valid for delivery; (b) a software system needs to be fully reworked after changes; and (c) the reuse of existing software artefacts is invalid.

Currently, software traceability is mandated by many standards (IEC 1999, UK_Ministry_of_Defence 1997) for software system development. According to the current literature, many approaches focus on putting software traceability into practice. A few works have been found investigating the costs and potential benefits of providing software traceability to the software development lifecycle. Examples of those works are (Murray et al. 2002, Ramesh et al. 1995b). The work in (Murray et al. 2002) has shown that software traceability appeals a low cost for reasonable benefit. The authors created a case study of three software systems and experienced the identification of relevant artefacts and generation of traceability relations between those artefacts. According to their experiments, it took 30 hours and two engineers for establishing traceability relations between 19 documents. The

traceability relations expressed information about a software system e.g. traceability relations indicated whether or not all requirements have been implemented, all design artefacts have been documented, and source codes have been implemented accurately. The authors concluded the cost of establishing traceability is affordable and the use of traceability relations is benefitial although they did not provide quantitative evidences of traceability benifits. Similarly, Ramesh et al. (Ramesh et al. 1995b) created a case study of traceability implementation. The authors discussed the cost and benefits of providing traceability in a software system. Traceability relations were used to decrease time and effort in the development lifecycle. According to their case study, software traceability was established with 60 work-months but dramatically decreased the budget of software system development. The authors discussed that establishment of software traceability increases workload and documentation. This leaded an initial budget of the software development higher; however, total lifecycle costs due to the development were significantly reduced. Nonetheless, the authors did not provide any quantitative study how much the cost was reduced.

Additionally, the use of software traceability tends to improve the software development process. Many researches in the current literature focus on investigating how to apply traceability relations effectively and practically. According to (Lindvall and Sandahl 1998), traceability relations are used for change-predicting and conducting details to support impact analysis. The details are used to increase the accuracy of cost estimation, and decrease time-consuming of analysis. In addition to (Boehm 2000, Boehm et al. 2004), the reuse of software artefacts to develop a new software system can cost around 15% of creating new artefacts. Traceability relations assist the reuse of software artefacts by conducting details of existing artefacts that are associated to new requirements. This concept has been raised in (Alexander 2003, CAFE 2003, Dick 1999, ESAPS, Knethen et al. 2002, Lindvall and K. 1996). The authors agreed applying of traceability relations increases the proportion of reuse and decreases the cost of developing a new system. We describe the following section the different use of traceability relations to support the activities in the software development process.

2.2.2. Different Use of Software Traceability

Software traceability can be used to assist with various activities in the development of software systems. Examples of these activities are (a) domain impact analysis, (b) validation, verification, and testing, (c) reuse of software artefacts, and (d) understanding software artefacts. In this section we describe some of the approaches that have been proposed to support those activities.

I. Domain Impact Analysis

The aim of domain impact analysis is to predict possible consequences of changes in the software artefacts (Lindvall and Sandahl 1998, Lock et al. 1999). Software traceability is a technique applied to support the activity of impact analysis. In particular, traceability relations can be used to identify artefacts that have been changed since traceability relations can associate relevant artefacts to a particular artefact being changed. Traceability relations are realised as *bi-directional* links that represent associations between two artefacts in both directions. In (Lindvall and Sandahl 1998), domain impact analysis is defined as analyzing which artefacts of a system are affected, and how the artefacts are affected, by a proposed change. When a requirement is proposed to be changed, all other artefacts associated to the requirement have to be changed to fulfill the new requirement. The authors suggested that traceability relations assist the estimation of the cost of a proposed change by considering which and how artefacts are affected. The authors also suggested that the use of traceability relations is required in different situations to support impact analysis of a proposed change. Examples of these situations are (i) straightforward tracing between a changed requirement to other artefacts i.e. requirements, design models, source code; (ii) tracing between textual references of a changed requirement to external documents; and (iii) tracing common names between artefacts regarding a changed requirement. In (Lock et al. 1999), the authors proposed to apply traceability techniques to support domain impact analysis. Their approach involves two activities namely: (i) *traceability extraction*, which generates traceability relations between artefacts for export to a database storage system; and (ii) *traceability analysis*, which analyses and represents possible impact of a

change using the database. However, in this work, the authors do not provide how to generate the traceability relations.

In (Richardson and Green 2003, Richardson and Green 2004), the authors proposed to use traceability relations to generate the effects of changes on source codes. The traceability relations, called *surface traceability*, are created between a program specification and the corresponding synthesized code. In these work, the synthesized codes are generated by applications namely, *AUTOFILTER* and *GNU C Compiler* (GCC). However, the authors only described that the technique used for the traceability generation is a lightweight technique. The small changes, called *perturbations,* are created on the specifications and the effects on the synthesized program are observed by using the traceability relations.

In (Cleland-Huang et al. 2005a), the authors proposed the *Goal Centric Traceability* (GCT) to support developers impact analysis on non-functional requirements. GCT is implemented through the four phases of: (a) *goal modeling*, which is an activity of modeling non-functional requirements with *softgoal interdependency graph*s (SIG); (b) *impact detection*, which traceability relations between functional model of a system and possible effects of a change are generated; (c) *goal analysis*, which the effect on whole system is developed according to the traceability relations created in previous phase; and (d) *decision making*, which stakeholders determine the impact results and take a decision regarding the change. In GCT, the generation of traceability relations is implemented by using a probabilistic network model proposed in (Wong and Yao 1991, Wong and Yao 1995).

Some approaches (Cleland-Huang et al. 2002b, Dhar and Jarke 1988, Knethen 2002b, Pinheiro and Goguen 1996, Ramesh and Dhar 1992) have claimed that their traceability techniques support domain impact analysis. In (Dhar and Jarke 1988), the authors described a knowledge based dependency learning and prediction mechanism regarding software artefacts in a system. In other words, traceability relations representing the dependencies illustrate an impact between software artefacts. The work is extended in (Ramesh and Dhar 1992). Ramesh and Dhar

defined an entity, namely *change_proposals* that appear in two sub-models of the traceability reference model for high-end traceability users *requirements management* and *design allocation sub-models*. The *change_proposals* entity is related to *system_objectives* and *requirements* entities with traceability relations called *generate* and *modify* in the *requirements management* sub-model while the entity is also related to *design* entity with traceability relations called *modify* in the *design allocation* sub-model. These types of traceability relations are used to maintain the history of updated artefacts in a system. Additionally, they defined an entity, namely *system_subsystems_components*, with traceability relations *depend_on* and *part_of*, used to support impact analysis. The authors suggested that the *depend_on* traceability relations can assist the maintenance of dependency information. The *part_of* traceability relations are used to discover dependencies within a particular artefact. In other words, the traceability relations are used to justify which artefacts may affect a particular changed artefact. When an artefact is created or modified, the traceability relations are used to assist in deciding the impact on the whole system.

In (Pinheiro and Goguen 1996), domain impact analysis is achieved by two different types of traceability relations: *replace* and *abandoned* relations. The *replace* relations are used to identify which artefacts are substituted by a new or updated requirement, while the *abandoned* relations are used to determine which artefacts become unused in a system due to a changed requirement. Cleland-Huang et al (Cleland-Huang et al. 2002b) proposed a traceability approach to support the process of analyzing the impact of a change with a concern on performance of the system by exercising existing traceability relations to determine possible effects. The approach also illustrates a comparison of the possible results to the original system. In (Knethen 2002a, Knethen 2002b), they described such an analysis. Egyed et al. (Egyed and Grunbacher 2002) described that their traceability technique can also help in analyzing the impact of a proposed change. When a proposed change is created, new scenario is created and executed with a running system. The approach creates a footprint reflecting the proposed change.

II. Validation, Verification, and Testing

The quality of software systems relies on how the systems satisfy users' needs. Traceability relations are used to identify associations between artefacts created during the development life-cycle. Finkelstein (Finkelstein 1991) pointed out a system whose requirements have not been represented using formal methods, an aid is required to achieve validation, verifying and testing of the requirements. According to (IEEE-830 1998), system development requires software traceability to assist validation, verification and testing the systems. In (Haumer et al. 2000), the authors proposed an approach of applying traceability relations to support the system development. The usage of a system is recorded and called *Real World Scenes* (RWS). RWS and other observational materials recorded by using video are structured as *Real World Example* (RWE). RWE reflect software artefacts i.e. requirements, design, program specifications in the system. The approach generates traceability relations between fragments of RWE, called *Real World Example Fragment*, and the software artefacts. It uses these traceability relations for validation the software artefacts in the current system against RWE. Murphy et al. (Murphy et al. 1995) presented an idea of formal specification language that supported system validation. They described using traceability relations to support consistency checking between requirements, designs and source codes. However, this work does not describe different types of traceability relations existing between artefacts.

Experiments in (Ramesh and Edwards 1993) described that stakeholders earn benefits from traceability relations in validation and verifying a system. For example, project managers specify artefacts e.g. project plans, constraints, rules, and policies and then assure that the specifications have been followed appropriately. Gotel et al (Gotel and Finkelstein 1995) initially proposed that pre-traceability activities in the system development, to assist activities of validation, verification and testing both formal and informal requirements recorded in different ways i.e. wish list, audio of meeting, meeting transcript, initial requirements specification, email message, revised requirements specification, and query. They suggested the generation of traceability relations that associate stakeholders and requirements and usage of these traceability

relations from requirements backward stakeholders. These assist traceability users to assure validation, verifying and testing the requirements. In (Spanoudakis et al. 1999), they are concerned with *overlaps* traceability relations. The work presents how the overlap traceability relations are used to detect inconsistency between requirements represented in a formal specification. Fiutem at el. (Fiutem and Antoniol 1998) suggested that traceability relations can ensure consistency between software artefact created during software development process. The authors present an approach for checking the fulfillment of object-oriented design with source code (implemented in C++). Checking is performed by applying traceability relations. The traceability relations called *as is* are generated between design artefacts and source code. These relations assist traceability users to deal with inconsistency between design and code. The work in (NASA) proposed the software development life-cycle and applied an activity called *traceability analysis* during development activities i.e. requirements analysis, design analysis, and implementation analysis. The work uses traceability relations to validate and verify the satisfaction of requirements, design and implementation.

In addition to, many traceability approaches (Letelier 2002, Pohl 1996b, Ramesh and Jarke 2001) are aimed to support the activity of validation, verification and testing a system. Pohl (Pohl 1996b) proposed the *process-centered requirements engineering* that enable traceability activity in the system development and support validation, verifying and testing the system. In (Ramesh and Jarke 2001), the authors proposed the traceability meta model containing traceability reference model to support the work in (Ramesh and Edwards 1993). In (Pinheiro 2000, Pinheiro and Goguen 1996), the authors claimed that their approach assisted the activities of validation, verification and testing a system such as: (i) *derive, refine, extract*, and *part of* traceability relations are used to describe the associations of requirements to other different artefacts i.e. design models and source code; (ii) *replace* and *abandon* traceability relations are used to identify unnecessary software artefacts existing during the development; and (iii) *test* traceability relations are used to associate requirements and test cases. Egyed (Egyed 2002) described two basic properties of traceability relations: (i) *bi-directionality* that means a traceability relation associates two software

artefacts in both two directions; and (ii) *transitivity* that two traceability relations relating three software artefacts can conclude another traceability relation between two of those software artefacts. They described tracing between software artefacts in a system by using those two properties in order to support validation and verification the system.

III. Reuse of Software Artefacts

There have been many approaches (Alexander 2003, Antoniol et al. 2002, CAFE 2003, Dick 1999, ESAPS, Knethen et al. 2002, Lindvall and K. 1996) to using traceability relations to assist with software system reuse. In (Knethen et al. 2002), the approach is proposed for recycling requirements in a system. The authors claimed that direct support for recycling requirements requires horizontal traceability relations. In this work, use case specifications and UML diagrams are used to represent requirements. The requirements are separately viewed as (a) *logical entity*, which is a textual part of requirements and (b) *documentation entity*, which is a software artefact i.e. functional requirement and design class. There are two types of models: (i) *conceptual system model* (CSM), which describes types of logical entities and their traceability relations between requirements; and (ii) *conceptual documentation model* (CDM), which describes types of documentation entities and their traceability relations between requirements. The CSM of new requirements and CDM of an existing system are developed. Traceability relations between logical and documentation entities are used to identify possible reused parts of existing artefacts.

The works in (Alexander 2003) illustrate the deployment of traceability relations in industrials. New requirements are compared with an existing system by using traceability relations between new and existing requirements. The traceability relations help with diagnosis of reuse existing artefacts. Antoniol et al. (Antoniol et al. 2002) proposed the generation of traceability relations from existing source code to new requirements represented in textual language. The authors suggested that the activity helps locating possible reused source code for satisfying the new

requirements. In (Lindvall and K. 1996), the authors suggested that traceability activity helps discovering existing software artefacts i.e. design and components that might be possible to reuse. Traceability relations are used for identification of such a design or component relating to a new requirement.

Recently, in the domain of product family systems, there have been some approaches that employed traceability activities in order to assist reuse. Dick (Dick 1999) suggested a *rich traceability* to enable supplemental artefacts for complying new requirements to existing requirements. The idea of work is to support the reuse of existing requirements for a new product member's requirements by adding extra requirements and removing unused requirements. The technique in this work is extended in (Hull et al. 2002). The authors proposed an artefact namely *satisfaction arguments* that are represented by using goal-structures charting AND/OR decompositions. The satisfaction argument contains domain knowledge, decision and issues artefacts in the domain of product family systems. The aim of the approach is to apply the rich traceability to support the reuse of the requirements for the product family system development.

Some projects (CAFE 2003, ESAPS) suggested traceability activities in the product family system development to support developers reuse of existing software artefacts i.e. requirements, design, software components, and source code to develop new product member. Traceability relations between new requirements of new product members and existing artefacts in product family systems are used to identify relevant existing software artefacts to the new product. The relevant software artefacts (i.e. design models and reusable software components) are then reused and integrated as a new product member in a product family.

IV. Understanding Software Artefacts

The works in (Ramesh and Edwards 1993, Ramesh et al. 1995) illustrated the observation of issues in the software development life-cycle. The authors concluded that stakeholders are required to understand software artefacts created during the

software development process and suggested that traceability relations can be used by different stakeholders to understand an artefact in different specifications. For example, software engineers need to understand requirements which are specified by different stakeholders e.g. project managers and requirements engineers, in order to implement the system.

In (Antoniol et al. 2002), they proposed to use traceability relations to comprehend existing source code. The traceability relations between source code and documents (i.e. handbook, design documents, and manual documents) support both *top-down* and *bottom-up* comprehension. In top-down comprehension, it provides a guideline on where to look for details of a particular artefact, while, in bottom-up comprehension, it provides the abstraction of artefacts. In (Maletic et al., Marcus and Meletic 2003), the authors proposed an approach for generating traceability relations between source codes artefacts with an emphasis on recognizing similarities between them.

The works in (Fairley and Thayer 1997, Haumer et al. 1998, Weidenhaupt et al. 1998) illustrated that traceability relations are required to support planning and controlling of projects. Traceability information is used to estimate and follow the plans. In (Haumer et al. 2000), traceability relations between existing artefacts are used to understand the satisfaction of requirements. In addition to (Dömges and Pohl 1998, Haumer et al. 1999), the authors suggested *rationale* traceability relations associated between software artefacts e.g. design decisions, alternatives, and assumptions in a system to support system comprehension. Gotel and Finkelstein (Gotel and Finkelstein 1994) suggested *contribution* traceability relations between stakeholders and requirements. The supplement attributes e.g. stakeholders' roles, system's policies, and constraints can be captured. These attributes in traceability relations are used by stakeholders to comprehend the requirements.

Software traceability can be used during the system development process in different activities. Many existing approaches for traceability generation have been proposed to achieve these activities. However, the deployment of traceability

relations can be achieved and be more advantageous, if the traceability relations are generated accurately, effectively and efficiently.

2.3. Problems with Software Traceability

Despite the importance of software traceability, establishing traceability relations is not an easy task and there are many associated problems. We describe below these problems.

I. The Difficulty to Manage Traceability in Large and Complex Systems due to Numerous Software Artefacts

Software artefacts generated during the software development life-cycle are numerous and diverse. Traceability activities, i.e. traceability generation, traceability visualization, and traceability usage, require dealing with those software artefacts. The traceability activities become more difficult when dealing with a large number of heterogeneous software artefacts.

II. The Lack of Automatic Support for Traceability

Manual traceability establishment is error-prone, difficult, time-consuming, expensive, complex, and limited on expressiveness. However, fully automatic support for traceability in software engineering environments and tools are not always adequate (Ramesh and Jarke 2001, Sherba et al. 2003b).

III. The Lack of Presenting of the Semantics of Traceability Relations for Specific-Domain Systems

In general, traceability relations have different meanings and relate to different types of artefacts. Many approaches (Alexander 2003, Dick 1999, Gotel and Finkelstein 1994, Knethen 2002a, Letelier 2002, Lindvall and Sandahl. 1996, Marcus and Meletic 2003, Pohl 1996b, Ramesh and Jarke 2001, Zisman et al. 2002b) proposed to classify different types of traceability relations based on different semantics aspect. However, the majority of these approaches do not focus on the classification

of traceability relations regarding some specific domains such as product family systems (We describe the literature of product family systems and traceability for these systems in Chapter 3).

IV. The Disjointed Process between Main Development Process and Traceability Activities

Some existing approaches (Antoniol et al. 2000, Gotel and Finkelstein 1995, Pinheiro and Goguen 1996, Pohl 1996b, Ramesh and Jarke 2001) proposed to establish traceability during the creation of software artefacts. However, the processes of software development and traceability are distinct. Stakeholders can be confused about the best time to do traceability activity. Therefore, it is necessary to support traceability activities after the artefacts have been created.

V. The Different Stakeholders' Needs

Different stakeholders require different traceability information with different perspectives and purposes of use. Software engineers need to justify if new requirements affect an existing system and require traceability relations to support impact analysis. Moreover, requirements engineers want to assure that all requirements have been achieved, and to support validation and verification of the system. Therefore, different traceability relations are needed to support different activities for different stakeholders.

VI. The Difficulty to Trace on Distributed Artefacts from Diverse Tools

There is a large number and heterogeneity of software artefacts in a system. The software artefacts can be created from different tools. It becomes difficulty to interoperate documents generated by different tools.

VII. The Fragility of Traceability Relations

Traceability relations are only beneficial to the development of software systems if they have been established properly and correctly. However, it is easy to break

traceability in a system. More specifically, if there is a change to a system that requires a change on some existing software artefacts, the existing traceability relations will need to be updated. Unfortunately, it is not easy to maintain update of traceability relations due to system's change.

2.4. Reference Models and Classification for Traceability Relations

Various reference models (Bayer and Widen 2002, CAFE 2003, ESAPS, Kim et al. 2005, Lago et al. 2004, Letelier 2002, Pohl 1996b, Ramesh and Jarke 2001, Toranzo and Castro 1999) and classification for traceability relations (Alexander 2003, Antoniol et al. 2002, Bayer and Widen 2002, CAFE 2003, Cleland-Huang et al. 2002a, Cleland-Huang et al. 2002b, Dick 1999, Egyed 2003, Egyed and Grunbacher 2002, Gotel and Finkelstein 1995, Hayes et al. 2003, Kim et al. 2005, Knethen 2002a, Lago et al. 2004, Lindvall and K. 1996, Maletic and Marcus 2001, Marcus and Meletic 2003, Pinheiro and Goguen 1996, Pohl 1996b, Ramesh and Jarke 2001, Sherba et al. 2003a, Spanoudakis et al. 2004, Zisman et al. 2002b) have been proposed in the literature. We describe below some of these approaches.

2.4.1. Reference Models

Pohl (Pohl 1994, Pohl 1996b) proposed a traceability meta model and four traceability reference models, namely *specification model*, *representation model*, *agreement model*, and *dependency model*. The specification model represents information i.e. the content of the specification independent of its representation according to some guidelines, standards, or domain models. The representation model represents information i.e. the various representation formats used during the software development process (e.g. natural language, graphical notations like ER diagrams or DFDs, formal notations like O-Telos). The agreement model represents information i.e. the different viewpoints of the stakeholders, arguments, alternative solutions, and decisions created during the software development process. The first three models represent traceability information while the *dependency model* defines types of traceability relations to be used to relate various software artefacts

according to the three models. The *dependency model* represents types of traceability relations in five different groups called: (a) *condition link*; (b) *content link*; (c) *documentation link*; (d) *evolutionary link*; and (e) *abstraction link group*. These five groups include 18 concrete types of traceability relations that are captured between software artefacts defined in the *specification*, *representation*, and *agreement* models. The condition links are traceability relations that relate restrictions to software artefacts. The content links are traceability relations that are concerned with the content of software artefacts. The document links are traceability relations that relate software artefacts to the source of the software artefacts. The evolutionary links are traceability relations that associate between different types of software artefacts created in different phases of software development. The abstraction links are traceability relations that represent the abstractions and concretion of software artefacts.

Based on experiences with the deployment of REMAP (Ramesh and Dhar 1992), Ramesh and Jarke (Ramesh and Jarke 2001) developed two traceability reference models for two groups of traceability users: (a) *low-end traceability users*, concerned with those users that have few years of experience in traceability and see the importance of traceability as a way of complying to standard; and (b) *high-end traceability users*, concerned with those users that have many years of experience in the area of traceability and see traceability as a way of guaranteeing customer satisfaction and knowledge creation through the system development life cycle. These two reference models, which one resembles the other, describe 50 types of traceability relations and 31 types of entities. The reference model for low-end traceability users contains seven types of traceability relations: *derive*, *developed_for, performed_on, depend_on, interface_with, allocated_to,* and *satisfy*. The reference model for high-end traceability users contains four sub-models: (i) *requirements management*; (ii) *rationale*; (iii) *design allocation*; and (iv) *compliance verification* sub-models, and contains 43 types of traceability relations. All together, the 50 types of traceability relations proposed in both high-end and low-end reference models are organized in four groups namely: (a) *satisfaction link*, which is used to ensure that a requirement is satisfied by a system; (b) *evolution link*, which is used to record the history of

documents (e.g. new, modified, and existing); (c) *rationale link*, which is used to represent the rationale behind the creation of a document; and (d) *dependency link*, which is used to represent dependencies between documents.

Additionally, Mohan and Ramesh (Mohan and Ramesh 2002) adapted the reference models from (Ramesh and Jarke 2001) for identifying common and variable requirements in the domain of product family systems. The adapted reference model includes primitives such as architectural decisions and design decisions that reflect the common and variable requirements in the product family system domain.

Toranzo and Castro (Toranzo and Castro 1999) described a reference model for different views of traceability users (i.e. requirements engineer, software engineers, and project engineers). However, the reference model presents types of traceability relations without any explicit semantics.

Letelier (Letelier 2002) proposed a reference model focusing on requirement specifications, namely *TraceableSpecification.* The specific types of TraceableSpecification are: (a) *RationaleSpecification*, which describes a rationale behind a TraceableSpecification, (b) *RequirementsSpecification*, which is a requirement or group of requirements, (c) *TestSpecification*, which define a test for a requirement, and (d) *OtherUML_Specification*, which is other type of UML models to elaborate the specification of a requirement. The reference model includes two main groups of traceability relations: The first group presents associations between two TraceableSpecifications by means of aggregation and contains relations called *part of.* The second group is composed of seven types of traceability relations: (a) *traceTo*, which represents a traceability relation between two TraceableSpecifications; (b) *rationaleOf*, which represents a traceability relation between a RationaleSpecification and TraceableSpecification; (c) *validatedBy*, which relates a RequirementsSpecification and TestSpecification; (d) *verifiedBy*, which represents that a TestSpecification verifies OthetUML_Specification; (e) *assignedTo*, which represents that OtherUML_Specification relates to RequirementsSpecification; (f) *modifies*, which represents the a stakeholder or group of stakeholders that modify a

TraceableSpecification; and (g) *responsibleOf*, which represents the stakeholder or group of stakeholders that are responsible for the definition and maintenance of a TraceableSpecification.

Bayer (Bayer and Widen 2001) defined a traceability reference model that consists of artefacts and relationships between these artefacts. The artefacts are created during three activities of product family system development, namely *scoping, architecture design,* and *implementation.* During the scoping activity, *feature* entity representing the requirements of whole product family and *product* entity representing product members of the family are created. The activity of architecture design is concerned with three views: *component view* consisting of *component* entity, *class view* consisting of *class* and *interface* entities, and *data structure view* consisting of *data* entity. During the activity of implementation, *code module, property*, and *property file* are created. The models contains relationships that can be categorized as three groups: (i) relations between different types of artefacts created in different activities, namely *realizes* and *implements*; (ii) relations between different types of artefacts created in the same activity, namely *has, accesses, implements, contains,* and *configures*; and (iii) relations between the same type of artefacts, namely *uses, excludes, depends on, aggregates, implements,* and *specializes.*

In (Kim et al. 2005), the authors defined a conceptual model, called *traceability map.* The model consists of artefacts created during the product family system development process and traceability relations. In this work, the artefacts are concerned with the creation during two sub-processes: *domain engineering* and *application engineering.* During the domain engineering sub-process, (a) the requirements artefacts are created and defined as *product line scope* representing the boundary of a product family, and *C&V model,* representing common and variable aspects in the family, and (b) the design and implementation artefacts are created and defined as *core asset.* During the application engineering sub-process, (a) the requirements artefacts are created and defined as *application analysis model* representing a conceptual analysis of a product member, and *application specification analysis model* representing a specification of a product member, (b) the design

artefacts are created and defined as *application specification design model* representing a design model of a product member, and *decision resolution model* representing design decision artefacts, and (c) the implementation artefacts are created and defined as *instantiated core asset* representing specific reused assets for a product member, *integrated application design,* representing the integration of a product member, and *application implementation*, representing implementation artefacts of a product member. The traceability relationships in the model can be grouped as: (a) *selected* and *refined* that associate between different requirements artefacts; (b) *realized, derived,* and *cluster* that associate between requirements and design artefacts; and (c) *resolved* that associates between design and implementation artefacts.

In (Lago et al. 2004), the authors defined a model, called *simplified representation model* that consists of artefacts and relationships in the domain of product family systems. The artefacts are grouped as: (a) *product family level*, composed of *product family feature model* and *product family feature*; a product family feature model represents a feature model for a family while a product family feature represents a single feature in a family; (b) *product level*, composed of *product feature model, product feature, product component map,* and *design decision*; these artefacts represent the requirements and design of a product member; and (c) *implementation level*, composed of *implementation assets* representing the implementation artefacts of a product family. The model illustrates two types of traceability relations: (a) between artefacts in different levels such as *supports* and *implements*; and (b) between artefacts in the same level such as *realizes*, *composeOf, requires* and *excludes*. In (Plankl and Bockle 2001), the authors also suggested the traceability reference model which focus the requirement artefacts in the domain of product family systems. The model consists of *derived,* which associates between the requirements of a product family and the requirements of a product member; *caused,* which associates between different requirements; and *is in version*, which associates the evolution of a requirement.

Additionally, in the projects (CAFE 2003, ESAPS), the meta model is suggested to represent two types of traceability relations. First the *vertical traceability relations* between different types of artefacts created in different phases of the development,

and second the *horizontal traceability relations* between (a) between the requirements of product family and the requirements of product members; and (b) between different versions of requirements. They defined the traceability meta-model to capture the relationships in the product family. However, the approach in (Coriat et al. 2000), applied during the activity of domain analysis in the projects includes the traceability activity. The authors suggested traceability relations to be grouped as (i) *is realized by, excluded*, and *included* between different requirements; and (ii) *is applied on*, between requirements and constraints.

2.4.2. Classification of Traceability Relations

In order to discuss the various types of traceability relations that have been proposed in the literature, we follow the classification given in (Spanoudakis and Zisman 2005). We describe below these traceability relations types and the types of software artefacts to which the traceability relations exist.

Dependency

Dependency traceability relations are relations that can be used to represent the reliance between artefacts in a system. A dependency relation may hold between

(a) *requirements and requirements specifications* – dependency traceability relations between different requirements have been proposed in (Alexander 2003, Knethen 2002a, Knethen et al. 2002, Maletic et al., Pohl 1996a, Ramesh and Jarke 2001, Spanoudakis et al. 2004, Zisman et al. 2002b). In (Knethen et al. 2002, Maletic et al.), the dependency traceability relations, are called *casual conformance*. In (Spanoudakis et al. 2004, Zisman et al. 2002b), dependency traceability relations are called *requires*. In (Bayer and Widen 2002), *depends_on* traceability relations associated between requirements called *features*.

(b) *requirements and design specifications* – dependency traceability relations between requirements and design have been suggested in (Gotel and Finkelstein 1995, Mohan and Ramesh 2002, Ramesh and Jarke 2001). In (Ramesh and Jarke 2001), dependency traceability relations are used to assist decision making and

system management. In (Mohan and Ramesh 2002), dependency traceability relations have been used to support activities of commonality and variability analysis in the domain of product family system. In (Gotel and Finkelstein 1995), dependency traceability relations are called *development relations.* These dependency relations are also used to represent relations between requirements and other types of software artefacts generated during the software development process.

(c) *requirements and scenarios, and implementation specifications* – Egyed (Egyed 2003) proposed dependency relations between requirements and scenarios and between requirements and source code to record the history of requirements generation and development of requirements.

Refinement

Refinement traceability relations associate two artefacts when an artefact describes more details of another artefact. Refinement relations are proposed in (Gotel and Finkelstein 1995, Knethen 2002a, Knethen et al. 2002, Mohan and Ramesh 2002, Pinheiro and Goguen 1996, Pohl 1996b, Ramesh and Dhar 1992, Sherba et al. 2003b). The refinement relations are called *generalizes/ specializes* relations in (Mohan and Ramesh 2002, Ramesh and Dhar 1992). They suggested a refinement relation to elaborate complexity of an artefact into the other artefact or a group of artefacts. In other words, an artefact specifies more details about the other artefact.

(a) *requirements and requirements specifications* – In (Gotel and Finkelstein 1995, Pohl 1996b), a refinement relation is referred as a traceability relation which holds between requirements and requirements. In (Pohl 1996b), they classified this type of traceability relations as *refines*, which is used to define that a requirements is elaborated in more details by the other requirement, and *generalises*, which is used to represent a generalization of a requirements or a group of requirements. In (Gotel and Finkelstein 1995), they called refinement traceability relations as *containment relations* and relate a requirement and a combination of refined requirements. In addition to (Bayer and Widen 2002), the authors defined refinement traceability relations, called *has*, between two types of requirements artefacts namely *feature* and *product.* Kim et al. (Kim et al. 2005) defined two

types of refinement relations between requirements artefacts in the domain of product family systems, called *selected* and *refined*. In (Lago et al. 2004), the authors suggested two types of refinement traceability relations called *supports* and *composedOf*. *Supports* relations represent the requirements of a product family fractioning to the requirements of product members while *composedOf* relations represent the mixture of the requirements of a product member. In (Plankl and Bockle 2001), the refinement relations between requirements are called *caused*.

(b) *requirements and design specifications* – In addition to (Kim et al. 2005), the authors defined refinement relations, namely *resolved*, associating between requirements and design artefacts in the domain of product family systems.

(c) *requirements and implementation specifications* – Bayer at el. (Bayer and Widen 2001) defined refinement traceability relations between requirements called *feature* and implementation artefacts called *property*.

(d) *design and design specifications* – Examples of approaches including refinement relations between design artefacts are (Jacobson 1992, Knethen 2002a, Mohan and Ramesh 2002, Pinheiro and Goguen 1996, Ramesh and Jarke 2001). In addition to (Bayer and Widen 2002), the authors suggested two types of refinement traceability relations, called *specializes* and *aggregates*, relating between same types of design artefacts i.e. components, class, interface, and data entities.

(e) *implementation and implementation specifications* – Bayer at el. (Bayer and Widen 2002) also defined two types of refinement traceability relations between different implementation artefacts as *contains* and *configures*.

Evolution

Evolution traceability relations are relations that denote software artefacts that have been changed, or an artefact that has been replaced by an other artefact during the development, maintenance, or evolution of the system. This type of relation has been suggested in (Gotel and Finkelstein 1995, Maletic et al. 2003, Pinheiro and Goguen 1996, Pohl 1996b, Ramesh and Dhar 1992, Ramesh and Jarke 2001). An evolution relation may hold between

(a) *requirement and requirement specifications* – In (Pohl 1996b), they defined five types of evolution relations. Three of the evolution relations represent association

between different requirements and are called: (i) *formalizes*, when a requirement is defined in a more formal way than another requirement specification; (ii) *satisfies*, when a requirement specification satisfies another requirement; and (iii) *replaces*, when a requirement has been substituted by another requirement specification. In (Pinheiro and Goguen 1996), they classified two types of evolution relations: (i) *replace*, when a requirements has been substituted by another requirement; and (ii) *abandon* when a requirements is not necessary anymore. In (Maletic et al.), they called this type of relations as *non-causal conformance*. In (Plankl and Bockle 2001), the authors defined evolution relations, namely *is in version*, which relate between the old and new versions of requirements.

(b) *requirement specifications and other types of artefacts generated in later phases of software development* – In (Gotel and Finkelstein 1995), evolution relations are called *temporal relations* and represent the history of requirements in different phases of development. In (Pohl 1996b), evolution relations are called *elaborates* relations and associate an artefact from later phases in the software development process to the particular requirement. In (Sherba et al. 2003b), evolution relations are named: (i) *allocated_by* relations that associate a requirements to analysis or design models; and (ii) *elaborated_by* relations that associate requirements to source code.

(c) *requirements and constraints specifications* – In (Pohl 1996b), one type of evolution relations has been proposed between requirements and constraints. This relation is called *based_on* and represents an association between a requirement and constraints that have influenced the generation of the requirement.

Conflict

Conflict traceability relations are relations that denote conflicting aspects between software artefacts in a system. This type of traceability relations has been proposed in (Alexander 2003, Pohl 1996b, Ramesh and Jarke 2001). It represents that an artefact has a context that may be opposed to another artefact. A conflict relation may hold between

(a) *requirement and requirement specifications* – In (Pohl 1996b), they defined two types of conflict traceability relations namely: (i) *conflicts* relations represent that a requirement has negative influence on another requirement; and (ii) *contradicts* relations represent an inconsistency between two requirements.

(b) *requirement specifications and other types of artefacts generated in later phases of software development* – In (Ramesh and Jarke 2001), they defined types of traceability relations and entities concerning the confliction. The entities are named: *decisions,* which describe decision information concerning the generation of requirements; *rationale*, which represents the rationale of requirements; *assumptions*, which describe conclusions assumed to be true; and *issues conflicts*, which represent the confliction between requirements. Conflict relations are named: (i) *resolve* relations, which associate between decision and issues conflicts entities; (ii) *affect* relations, which relate between requirements and decisions; (iii) *generates* relations, which associate between requirements and issues conflicts; and (iv) *based on* relations, which relate between requirements and rational entities, between assumptions and rationale entities, and between rationale and decisions entities.

Overlap

Overlap traceability relations are relations that associate two artefacts referring to common aspects of a system. An overlap relation may hold between

(a) *requirements and requirements specifications* – Many approaches proposed overlap relations between different requirements such as (Antoniol et al. 2002, Egyed 2002, Gotel and Finkelstein 1995, Hayes et al. 2003, Jarke 1998, Knethen 2002b, Knethen et al. 2002, Pohl 1996b, Ramesh and Jarke 2001, Sherba et al. 2003a, Spanoudakis and Finkelstein 1997, Spanoudakis et al. 1999, Spanoudakis et al. 2004, Zisman et al. 2002b). In (Egyed 2002), they defined traceability relations called *commonality* that associate two artefacts with a common aspect. In (Gotel and Finkelstein 1995), they proposed overlap relations called *adopts* relations between different requirements. In (Knethen 2002a, Knethen et al. 2002), they defined overlap relations as *representation* relations and show an association between two artefacts with the same aspect of requirements.

(b) *requirements specifications and source of requirements that influence the generation of requirements* – In (Pohl 1996b), they defined two types of overlap relations namely: (i) *example_for* relation that relates between requirements and scenarios; (ii) *purpose* relation that relates between a requirement and purpose of the requirement; (iii) *background* relation that relates between a requirement and background of the requirement; and (iv) *comment* relation that relates between a requirement and comment regarding to the requirement.

(c) *requirements specifications and other types of artefacts in later phases of software development* – In (Spanoudakis et al. 2004, Zisman et al. 2002b), overlap relations are defined between requirements and object models. In (Pohl 1996b), they defined overlap relations called *Test_Case_for* relations that relate a requirement and test case. In (Antoniol et al. 2002, Marcus and Meletic 2003), they proposed overlap traceability relations between requirements and source code. In (Egyed and Grunbacher 2002), they suggested that overlap relations represent commonality between source code and design models.

(d) *scenarios and design specifications* – In (Egyed 2003, Egyed and Grunbacher 2002), they defined overlap relations between scenarios and design models i.e. class diagrams, use case diagrams, and data flows.

(e) *design and design specifications* – Overlap relations between different design artefacts are proposed in (Bayer and Widen 2002, Knethen 2002a, Knethen 2002b). More specifically, in (Bayer and Widen 2001), the authors suggested overlap traceability relations associating between component and class entities created during architecture design.

Satisfiability

Satisfiability traceability relations are relations that show how a system satisfies the requirements. A satisfiability relation may hold between

(a) *requirements and design specifications* – Satisfiability relations between requirements and design artefacts are proposed in (CORE, Ramesh and Jarke 2001, Zisman et al. 2002b); In (Zisman et al. 2002b), satisfiability relations are called *realise* relations. In (Bayer and Widen 2001), the authors defined satisfiability traceability relations, called *realizes,* relate the feature entities of the requirement

artefacts and the component entities of architecture artefacts. Kim et al. (Kim et al. 2005) defined two types of satisfiability relations between requirements and design artefacts in the domain of product family systems, called *realized* and *derived.* In (Lago et al. 2004), the authors also defined satisfiability traceability relations, called *realizes,* which associate between requirements of a product member and design decision artefacts.

(b) *requirements and implementation specifications* – In (Sherba et al. 2003b), satisfiability traceability relations between a requirement and source code are named *implemented_by.*

(c) *requirements and constraints specifications* – Satisfiability relations between a requirement and constraints are proposed in (Gotel and Finkelstein 1995, Pohl 1996b, Ramesh and Jarke 2001).

(d) *requirements and requirements specifications* – Satisfiability relations between different requirements are proposed in (Alexander 2003, Coriat et al. 2000, Dick 1999, Pinheiro and Goguen 1996, Plankl and Bockle 2001). In (Pinheiro and Goguen 1996), satisfiability relations are called *derive* relations and illustrate that a requirement is derived from another requirement. In other words, when a requirement is satisfied, its derived requirements should also be satisfied. This does not necessarily mean true in vice versa. In (Dick 1999), satisfiability traceability relations are named: *establishes* and *contributes* relations. In (Coriat et al. 2000), the satisfiability relations called as *is realized by* associate between non-functional requirements and functional requirements. In (Plankl and Bockle 2001), the authors defined satisfiability relations, namely *derived,* which relate between the requirements of product family and the requirements of product members.

(e) *design and implementation specifications* – In (Sherba et al. 2003b), satisfiability traceability relations are called *implemented_by* and are used to relate between design artefacts and source code. In addition to (Bayer and Widen), satisfiability traceability relations, called *implements,* relate between class and interface entities of architecture artefacts and code module entities of implementation artefacts. Kim et al. (Kim et al. 2005) also defined satisfiability relations, namely *resolved,* between design and implementation artefacts in the domain of product family

systems. According to (Lago et al. 2004), the authors defined satisfiability traceability relations, called *implements,* which associate between design decision artefacts and implementation asset artefact.

(f) *design and design specifications* – In (Bayer and Widen 2001), three types of satisfiability traceability relations: (i) *uses,* which associates between different component entities; (ii) *accesses*, which associates between component entities and data entities; and (iii) *implements,* which associates between class entities and data entities.

Rationale

Rationale traceability relations are relations that are used to connect software artefacts concerned with decisions and arguments. Rationale traceability relations can be found between: (i) different types of artefacts created during software development and the rationale specifications; and (ii) same types of artefacts. For the latter case, the rationale artefacts are included in the artefacts that are concerned with the rationale. Thus, a rationale relation may hold between:

(a) *rationale artefacts and other types of artefacts* – In (Ramesh and Jarke 2001), one sub-model of traceability reference model for high-end traceability users is called a *rationale* sub-model. The rationale sub-model has entities namely *object* that can be *software components*, *requirements* and *designs*, *rationale*, *decisions*, *issues_or_conflicts*, *alternatives*, *decisions*, *assumptions*, *arguments*, and *critical success factors* (CSF). The types of traceability relations in the sub-model are grouped as rationale relations and named: *based_on, affect, generate, address, influence,* and *depend_on.* In (Letelier 2002), rationale relations are called *rationelOf* relations and associate between *RationaleSpecification* and *RequirementSpecification, TestSpecification*, and *OtherUML_Specifications.* Additionally, In (Pohl 1996b), rationale relations are defined in a group called *condition link* which consists of two types of relations namely: (i) *precondition* that is used to relate a condition to a requirement which must be fulfilled to enable an implementation of the requirement; and (ii) *constraint* that is used to relate a constraint to a particular software artefact created in later phases of software development. In (Coriat et al. 2000), rationale

relations, called *is applied on,* associate between requirements and constraints specifications.

(b) *requirements and requirements specifications* – In (Bayer and Widen 2001), the authors defined rationale relations, called *excludes,* that relate between different features. Lago et al. (Lago et al. 2004) defined two types of rationale relations, called *requires* and *excludes,* which associate between the requirements of a product family. In (Coriat et al. 2000), two types of rationale relations, namely *excludes* and *includes by* are related between different requirements to represent the prohibitions and constitution between two requirements, respectively.

Contribution

Contribution traceability relations are relations that denote a stakeholder or group of stakeholders who have contributed to the generation of a software artefact. Gotel and Finkelstein (Gotel and Finkelstein 1995) initially proposed this type of traceability relations and called the relations as *pre-traceability.* In (Sherba et al. 2003b), two types of contribution relations are named: (i) *discussed_by* relations that associate between a requirements and a stakeholder or group of stakeholders who has contributed to the generation of the requirement; and (ii) *elaborated_by* relations that associate a stakeholder or group of stakeholders who are contributed to source code.

Table 2-1 shows a summary of different types of traceability relations that are proposed by different approaches. We present the types of traceability relations by classifying the generation of traceability relations based upon different types of software artefacts i.e. requirements, design, source code, source of requirements, constraints, test cases, and rationale.

Table 2- 1: Different traceability relationships between different artefacts

Software artefacts	Traceability relations
Between **requirements specifications** and **requirements specifications**	- **dependency** (Alexander 2003, Bayer and Widen 2002, Egyed and Grunbacher 2002, Gotel and Finkelstein 1995, Knethen 2002a, Knethen et al. 2002, Maletic et al., Pohl 1996a, Ramesh and Jarke 2001, Spanoudakis et al. 2004, Zisman et al. 2002b) - **refinement** (Bayer and Widen 2002, Gotel and Finkelstein 1995, Kim et al. 2005, Knethen 2002a, Knethen et al. 2002, Lago et al. 2004, Mohan and Ramesh 2002, Pinheiro and Goguen 1996, Plankl and Bockle 2001, Pohl 1996a, Ramesh and Dhar 1992, Ramesh and Jarke 2001, Zisman et al. 2002a) - **evolution** (Gotel and Finkelstein 1995, Maletic et al., Pinheiro and Goguen 1996, Plankl and Bockle 2001, Pohl 1996a, Ramesh and Dhar 1992, Ramesh and Jarke 2001) - **conflict** (Pohl 1996a, Ramesh and Jarke 2001). - **overlap** (Antoniol et al. 2002, Egyed 2002, Egyed and Grunbacher 2002, Gotel and Finkelstein 1995, Hayes et al. 2003, Jarke 1998, Knethen 2002b, Knethen et al. 2002, Pohl 1996b, Ramesh and Jarke 2001, Sherba et al. 2003a, Spanoudakis and Finkelstein 1997, Spanoudakis et al. 1999, Spanoudakis et al. 2004, Zisman et al. 2002b) - **satisfiability** (Alexander 2003, Coriat et al. 2000, Dick 1999, Pinheiro and Goguen 1996, Plankl and Bockle 2001)
Between **requirements specifications** and. **design specifications**	- **dependency** (Gotel and Finkelstein 1995, Mohan and Ramesh 2002, Ramesh and Jarke 2001). - **satisfiability** (Bayer and Widen 2002, CORE, Kim et al. 2005, Lago et al. 2004, Ramesh and Jarke 2001, Spanoudakis et al. 2004, Zisman et al. 2002b). - **conflict** (Ramesh and Jarke 2001). - **overlap** (Knethen 2002a, Knethen 2002b, Spanoudakis et al. 2004, Zisman et al. 2002a, Zisman et al. 2002b) - **refinement** (Jacobson 1992, Kim et al. 2005, Mohan and

	Ramesh 2002, Pinheiro and Goguen 1996, Ramesh and Jarke 2001) - **evolution** (Gotel and Finkelstein 1995, Pohl 1996b, Sherba et al. 2003a)
Between **design specifications** and. **design specifications**	- **dependency** (Jacobson 1992, Knethen 2002a, Ramesh and Jarke 2001) - **refinement** (Bayer and Widen 2002, Jacobson 1992, Knethen 2002a, Mohan and Ramesh 2002, Pinheiro and Goguen 1996, Ramesh and Jarke 2001) - **overlap** (Bayer and Widen 2002, Knethen 2002a, Knethen 2002b) - **satisfiability** (Bayer and Widen 2002)
Between **design specifications** and **implementation specifications**	- **satisfiability** (Bayer and Widen 2002, Kim et al. 2005, Lago et al. 2004, Sherba et al. 2003b) - **overlap** (Egyed and Grunbacher 2002)
Between **implementation specifications** and **implementation specifications**	- **refinement** (Bayer and Widen 2002)
Between **design specifications** and **scenario specifications**	- **overlap** (Egyed 2003, Egyed and Grunbacher 2002)
Between **requirements specifications** and **implementation specifications**	- **dependency** (Egyed 2002, Egyed and Grunbacher 2003) - **overlap** (Antoniol et al. 2002, Marcus and Meletic 2003) - **satisfiability** (Sherba et al. 2003a) - **refinement** (Bayer and Widen 2002)
Between **requirements specifications** and	- **dependency** (Egyed 2003)

scenarios specifications	
Between **requirements specifications** and **source of requirements**	- **overlap** (Pohl 1996b) - **contribution** (Gotel and Finkelstein 1995, Sherba et al. 2003a)
Between **requirements specifications** and **constraints specifications**	- **evolution** (Pohl 1996b) - **satisfiability** (Gotel and Finkelstein 1995, Pohl 1996b, Ramesh and Jarke 2001)
Between **requirements specifications** and **test case specifications**	- **overlap** (Pohl 1996b)
Between **implementation specifications** and **source of requirements**	- **contribution** (Sherba et al. 2003b)
Between **rationale specifications** and **requirements specifications**	- **rationale** (Letelier 2002, Pohl 1996b, Ramesh and Jarke 2001)
Between **rationale specifications** and **design specifications**	- **rationale** (Letelier 2002, Pohl 1996b, Ramesh and Jarke 2001)
Between **rationale specifications** and **implementation specifications**	- **rationale** (Ramesh and Jarke 2001)
Between **rationale specifications** and	- **rationale** (Letelier 2002)

test case specifications	
Between **rationale in requirements specifications** and **rationale in requirements specifications**	- **rationale** (Bayer and Widen 2002, Coriat et al. 2000, Lago et al. 2004)

2.5. Approaches for Establishing Traceability Relations

In this section we describe existing approaches to support generation of traceability relations. These approaches can be classified into three groups depending on their level of automation namely (a) manual approaches, when the traceability relations are generated manually by the users with or without the support of traceability tools; (b) semi-automatic approaches, when the traceability relations are generated based on the existence of previously manually defined relations; and (c) automatic approaches, when the traceability relations are generated without human interaction. We describe below these approaches.

2.5.1. Manual Establishment of Traceability Relations

As mentioned earlier, some existing approaches (Dorfman and Flynn 1984, Han 2001, Kaindl 1992, Watkins and Neal 1994) and commercial tools (CaliberRM, DOORS, RDT, RequisitePro, RTM, TestDirector) assume the establishment of traceability relations to be manual in which the users are supposed to specify the elements in the documents to be traced and even the types of traceability relations associated with these elements.

Although some of these approaches offer tool support to assist with the activity of traceability generation. This support is mainly concerned with the display of the various documents and elements to be traced, the selection of the elements to be traced, the selection of the different types of traceability relations, and the visualisation of the traceability relations. In (Kaindl 1992), the author developed a tool that allows traceability users to create and visualise traceability relations between software artefacts. The work is intended to support software artefacts which are specified in natural language and uses hypertext technique for representing and visualising traceability relations between software artefacts. However, the identification of traceability relations is a manual effort. In (Dorfman and Flynn 1984), they developed a tool, called ARTS for supporting traceability generation and visualisation. The tool allows traceability users to define the

templates of software artefacts, specify the software artefacts according to the templates, manually create traceability relations between the software artefacts, and visualise the traceability relations by means of reports and queries. In (Watkins and Neal 1994), they provided a tool, called ATS. The tool allows users to manually identify traceability relations between data in a database and visualise traceability relations by mean of pre-defined reports. In (Han 2001), the author proposed a traceability reference model and developed a tool, called TRAM for traceability generation. He also defined a set of templates for specifying two types of software artefacts i.e. requirements and software architecture in a system. The tool allows traceability users to manually create traceability relations between requirements and software architecture according to the traceability reference model.

Moreover, some existing commercial requirement management (RM) tools have been proposed to support traceability activities, particularly traceability generation, such as CaliberRM (CaliberRM), DOORS (DOORS), RDT (RDT), RTM (RTM), RequisitePro (RequisitePro), and TestDirector (TestDirector). DOORS (DOORS) is a requirements management tool developed by Telelogic that provides a Microsoft Explorer-like and spreadsheet-like interfaces for navigating and displaying documents. DOORS provides functionalities for capturing, tracing, and managing software artefacts in a system. One of the major features of DOORS is its ability to create relations between software artefacts generated by the tool. The idea is that the tool allows users to establish traceability relations between software artefacts after the artefacts are created. However, the tool only supports the creation of traceability relations for the software artefacts that have been created by the tool.

Other approaches (Alexander 2003, Dick 1999) have extended DOORS (DOORS) for capturing and recording traceability relations. The approach in (Dick 1999) use *rich traceability* technique which define traceability relations between requirements, and other different artefacts called *satisfaction arguments*. Satisfaction arguments include domain knowledge, and decision and issues of a particular requirements, and are represented by goal-structures charting, and AND/OR decompositions in DOORS. In this work the traceability relations are manually established by using

DOOR. In (Alexander 2003), the author proposed an approach that apply the *Volere* template (Volere) for generating software artefacts, DOORS for capturing traceability information, and hypertext techniques for representing traceability information.

In (Ramesh and Jarke 2001), the authors defined a traceability meta model in which traceability reference models are described. They applied experiences from applying a conceptual model called *REMAP* (Ramesh and Dhar 1992). The traceability meta model is used as a language for defining different artefacts i.e. issues, arguments, assumptions, decisions, constraints requirements, and design. This work is applied with the Rationale Capture part of Andersen Consulting[1]' *Knowledge Based Software Assistant* (KBSA) *ADM* tool. The tool provides functionalities: (a) creation of artefacts according to the traceability reference models; (b) generation of traceability relations between artefacts created by the tool; and (c) visualization the traceability relations. However, those activities are performed by stakeholders.

Some approaches (Gotel and Finkelstein 1995, Haumer et al. 2000, Jarke 1998, Kotonya and Sommerville 1998, Letelier 2002, Sutcliffe and Maiden 1998) proposed the frameworks, techniques, and approaches of traceability in system development. In (Gotel and Finkelstein 1995), the approach describes traceability establishment between different requirements and between stakeholders and requirements; however, it does not state a support for automatic generation of those traceability relations. It is assumed that the activity should be concerned and done manually.

Some approaches are proposed for traceability activities in the domain of product family systems. Examples of these approaches are (Bayer and Widen 2002, Berg and Bishop 2005, CAFE 2003, Coriat et al. 2000, ESAPS, Kim et al. 2005, Lago et al. 2004, Mohan and Ramesh 2002, Plankl and Bockle 2001, Riebisch and Philippow 2001). However, the activities are assumed to be done manually. The authors do not explicitly define how to achieve in an automatic way or provide tool support for the

[1] The company has been currently called *accenture*.

traceability generation activity. An exception is found in (Kim et al. 2005), where the authors suggested to use rules for traceability generation. However, they do not define how to apply the rules in an automatic way.

2.5.2. Semi-Automatic Establishment of Traceability Relations

Some existing approaches (Cleland-Huang et al. 2002b, Egyed and Grunbacher 2002, Pinheiro and Goguen 1996, Pohl 1996b) are aimed to support an automatic traceability generation. However, the approaches are considered as semi-automatic since they require some manual efforts from traceability users such as specification of artefacts to be traced or identification of the types of traceability relations. We classify those approaches into four types depending on the techniques used. The four types include, (a) process-centered techniques, (b) event-based techniques, (c) scenario-based techniques, and (c) axiom-based techniques as described below.

Process-centered technique:

Pohl (Pohl 1996b) proposed an approach, called PRO-ART, to support generation and visualization of pre-traceability under a process-centered engineering environment. The approach depends on *Requirements Engineering Environment* (REE) in which requirements are represented as hypertext models (Pohl 1996a), extended entity-relationship models (Pohl and Haumer 1995), structured analysis models (Pohl 1996b), object models and behavior models (OMT), and O-Telos (Pohl 1996b). The approach provides integrated tools to support various activities in the environment. Examples of these activities are (i) execution of the software development process; (ii) capturing of traceability relations between different artefacts and between artefacts and stakeholders during the software development process; and (iii) visualization of traceability relations. Pohl also proposed a process repository to record the executed processes and traceability relations. The traceability relations can be generated manually and automatically according to the concrete traceability reference model for a specific system. However, the concrete traceability reference model must be manually defined. In other words, stakeholders

need to specify the structure of traceability relationships between artefacts being created during the development process.

Event-based technique:

In (Cleland-Huang et al. 2002b), they defined an event-based traceability framework to support tracing different requirements. The approach particularly supports impact analysis on both functional and non-functional requirements which are represented in natural language. A change on requirements (i.e. new, updated, deleted and abandon requirement) drives an action of traceability generation between requirements. The traceability relations are specified at the level of document entities. In this work, a prototype tool was developed to support generation of traceability relations. An automatic control namely *event manager* can automatically respond to an event of a change and enable activities of traceability generation i.e. generate new traceability relations and update existing traceability relations. However, some events such as updating existing traceability relations require traceability users to manually create the events.

Scenario-based technique:

Egyed and Grunbacher (Egyed and Grunbacher 2002) proposed a scenario-based approach for traceability generation which is extended from (Egyed 2001). In (Egyed 2003, Egyed and Grunbacher 2002), they described a prototype tool which is claimed to automatically generate traceability relations between model elements (i.e. use case diagrams, class diagrams and data flow diagrams), source codes, and scenarios (i.e. test case scenarios, usage scenarios). However, the approach requires an initial manual process for creating pre-defined traceability relations, called *hypothesized traces*. These hypothesized traces are identified between model elements and scenarios and are used in a second step to automatically create new traceability relations based on traceability rules and transitivity of the hypothesized traces. The approach uses scenarios to discover associations while a system is running. An association between a scenario and a particular artefact is called a *footprint*. A set of footprints and hypothesized traces are then recognised as a *footprint graph*. The

footprint graph is represented for traceability relations between an artefact and scenario.

Axiom-based techniques:

Pinheiro and Goguen (Pinheiro 2000, Pinheiro and Goguen 1996) proposed an approach for traceability generation that uses axiom techniques. In (Pinheiro and Goguen 1996), *Traceability of Object-Oriented Requirements* (TOOR) is created to provide: (i) *project specification* functionality, which allows traceability users to specify templates of artefacts (i.e. requirements, design, and source code) and define a structure of traceability relations; (ii) a functionality for instantiating artefacts; (iii) a functionality for creating traceability relations between different artefacts and between artefacts and stakeholders; and (iv) a functionality for visualising traceability relations. The specification of templates is applied with *Functional and Object-Oriented Programming Systems* (FOOPS) (Socorro 1993). Traceability relations are generated based on axiom techniques. Artefacts are recognized as operands and applied with logical operators in axioms. Traceability relations are identified when the tool analyse the implications of axioms. The visualisation of traceability relations is provided in three ways namely: *selective*, which allows traceability users to visualise traceability relations according to a specific query; *interactive*, which allows traceability users to query related artefacts according to a particular artefact; and *nonguided*, which allows traceability users to visualize all traceability relations. However, specifying the templates of artefacts, defining the structure of traceability relations, and instantiating the artefacts must be done by traceability users.

Those approaches have attempted to enable automatic support for establishing traceability relations. However, some of activities during the traceability generation process such as defining types of traceability relations to be created or identifying types of software artefacts to be traced are still performed by manual.

2.5.3. Fully Automatic Establishment of Traceability Relations

We classify the different types of approaches that support the generation of traceability relations in a fully automatic way into three types depending on the techniques used to assist with this task. These three types include, (a) information retrieval techniques, (b) rule-based techniques, and (c) hypermedia and information integration techniques as described below.

Information Retrieval (IR) techniques:

Antoniol et al. (Antoniol et al. 2002) applied IR techniques to generate traceability relations between source code documents represented in C++ and Java and requirements specified in natural language. Their approach uses both a *probabilistic method* and *vector space* model. It consists of using comments and identifier names within the source code to find similarities in the documents. The documents are ranked by relevance. Then the traceability relations are created based on the relevance of the documents. The work has been experimented with two case studies namely *LEDA* and *Albergate*. The experimental results have demonstrated high percent of recall measurements; however, fairly low percent of precision measurements. The authors described that both two models achieve almost the same recall measurements. However, the vector space model returns regular recall measurements with different numbers of documents in the experiments.

In (Maletic et al., Marcus and Meletic 2003), the authors proposed to use *Latent Semantic Indexing* (LSI) for establishing traceability relations between source code and other different types of documents such as requirements, designs, and test cases. The authors argued that their approach achieved better results than (Antoniol et al. 2002) in terms of recall and precision measurements. This approach requires full parsed source code and analysis of documents. It takes into consideration synonyms of context in documents. Marcus et al. argue that their approach also requires less processing of the source code and documentation and is language, programming language, and paradigm independent.

Hayes et al. (Hayes et al. 2003) proposed to use IR techniques to improve traceability generation. In particular, the approach applied three vector space IR techniques: (i) *vanilla vector retrieval*, which is a classical vector IR model for information retrieval; (ii) *retrieval with key-phrases*, which is an extension of the classical vector IR model that associates a list of key-phrases with documents and develops possible relevant phrases to match between documents; and (iii) *thesaurus retrieval*, which is an extension of the classical vector IR model that constructs a thesaurus and then associates the thesaurus with vocabulary in documents. In (Hayes et al. 2003), the approach (i) parses requirements as tokens, (ii) ignores unnecessary words which are not considered for matching (e.g. shall, the, for, etc.), (iii) constructs a list of tokens and thesaurus; and (iv) develops associations of documents. According to their experiments, the approach achieves better recall measurements but lower precision measurements when compared to classical IR techniques. This work has been later supported by a tool, called *RETRO* (Hayes et al. 2004). The authors have demonstrated in the latter work that the tool can facilitate the automatic traceability generation with reasonable recall and precision measurements.

However, the generation of traceability relations with the IR techniques does not take into consideration the semantic of artefacts being compared.

Rule-based approaches:

In (Spanoudakis et al. 2004), a rule-based approach to support generation of traceability relations has been proposed. The approach generates traceability relations between different types of requirement documents i.e. *customer requirements specification* (CRS), *functional requirements specification* (FRS), and object model. In the approach, traceability rules take into consideration the grammatical roles of the terms used to specify requirements in CRS and FRS. The approach is based on XML in which both documents and traceability rules are represented in XML-format. Initial experiments have demonstrated 52-94 percent of precision and 46-95 percent of recall measurements.

The generation of traceability relations by using rule-based approaches enables the consideration of the semantics of documents being compared and the traceability relation.

Hypermedia and information integrators:

Sherba et al. (Sherba et al. 2003b) proposed an approach that applies hypermedia and information integration techniques for supporting traceability activities such as traceability generation and visualisation. This work uses techniques of information integrators, called *Infinite* proposed in (Anderson et al. 2002), and open hypermedia. The approach applies Infinite integrators for creating explicit relations and anchors in documents. An anchor is an interested element in a document. Then, the approach discovers implicit traceability relations between documents by using created anchors and explicit relations. The authors described that the creation of the anchors depends on the algorithms used by the integrators. The algorithms used by the integrators can be IR techniques or runtime analysis. Thus, the algorithms can be as simple as a keyword search or as complex as a LSI technique. The tool called TraceM has been developed and provides the following functionalities: (i) *registration*, which allows users to register artefacts generated by heterogeneous tools, types of traceability relations, and new translators and integrators for supporting new types of artefacts; (ii) *scheduling*, which is used to schedule the execution of translators and integrators; (iii) *relationship mapping*, which allows users to generate traceability relations between documents; (iv) *evolution*, which allows users to update traceability relations; (v) *query*, which allows users to set an inquiry about information based upon existing documents and traceability relations; and (vi) *export*, which allows users to visualise a summary of traceability relations according to a particular artefact in HTML format. Since the tool is aimed to support various tools, the approach does not depend on particular specification and programming languages. However, the approach depends on the integrators and their applied algorithms and do not provide the evaluation by means of precision and recall measurements.

The majority of existing approaches do not support generation of traceability relations in a fully automatic way, although some of these approaches have

attempted to achieve a fully automatic generation of traceability relations. Moreover, some of semi- and fully- automatic approaches do not demonstrate reasonably experimental results and none of them are provided to support domain-specific systems such as product family systems.

2.6. Representation, Recording, and Maintenance of Traceability Relations

Since the majority of existing approaches for traceability generation do not support a fully automatic generation, basic techniques i.e. *identifier*, *tagging*, *indexing*, and *table* are used to assist representation of traceability information. These representation techniques can be done by manual or automatic. Some advanced techniques i.e. *mark-up, mapping graph,* and *hyperlink* are also used to represent traceability relations. Additionally, techniques i.e. database and special repositories are used for recording traceability relations. We describe below techniques which support activities in representing, recording, and maintenance of traceability relations.

2.6.1. Identifier Technique

Identifier technique uses a unique number to identify an artefact in a system and runs the number to other relevant artefacts (Sawyer et al. 1993, Sommerville 2001). In other words, identifier technique requires identifier numbers (ID no.) being created for each artefact. Traceability relations can be captured between identifiers. An identifier can be composed of a unique name and supplement information i.e. the name of a system and a type of an artefact. Figure 2-1 shows an example of identifier technique. The identifier is constructed from three dash-separated parts: (i) abbreviation of a system which a requirement belongs to e.g. mobile-phone (MP); (ii) abbreviation of an artefact type e.g. user interface (UI), design model (DM), source code (SC); and (iii) ordering number of a requirement e.g. 1, 1.1. As shown in Figure 2-1, a requirement MP-UI-1.1 is refined a requirement MP-UI-1.

MP-UI-1 The user interface shall have graphical menu.
MP-UI-1.1 The screen shall provide a list of graphical icons whose represents a menu and a background.

Figure 2- 1: Representing traceability by using identifiers

This technique supports both manual and automatic approaches for generating traceability relations and is best suited for capturing one-to-one or one-to-many vertical traceability relations e.g. between two requirements specifications. However, it is difficult to apply the technique to represent many-to-many and the technique represents traceability relations without their semantics. Using identifier technique is found in existing traceability approaches and tools such as (Alexander 2003, Dick 1999, DOORS, RequisitePro, RTM). Identifier technique has been extended by other techniques to represent additional traceability information. We describe below the extension of the technique.

2.6.2. Tagging Technique

Tagging technique uses added information to represent the semantic of traceability relations in artefact specification (Sawyer et al. 1993, Sommerville 2001). As shown in Figure 2-2, a requirement *MP-UI-1* has an attribute namely *Source* representing who created the requirement. The technique is suited for capturing one-to-one, one-to-many, or many-to-many relations and supports representing traceability relations with their semantics. However, the use of this technique may make the artefact specifications harder to read.

MP-UI-1 The user interface shall have graphical menu.
Source: product manager A
MP-UI-1.1 The screen shall provide a list of graphical icons whose represents a menu and a background.

Figure 2- 2: Representing traceability by tagging attributes

2.6.3. Indexing Technique

Indexing technique arranges artefact specifications into a group (Kotonya and Sommerville 1998, Sommerville and Sawyer 1997). The artefacts are identified by

using identifiers. As shown in Figure 2-3, requirements *user interface of screen display* are specified with identifiers *MP-UI-1* to *MP-UI-9*. The technique is suited for representing one-to-one and one-to-many traceability relations and traceability relations are only readable in one direction. However, the technique represents traceability relations without their semantics and it is hard to read in another direction and not practical with many-to-many relations.

No.	Requirements
MP-UI-1 – MP-UI-9	User interface of screen display
MP-UI-10 – MP-UI-15	User interface of embedded application i.e. games, clock, calendar.
MP-UI-16, MP-UI-27	User interface of network connection

Figure 2- 3: Representing traceability by indexing

2.6.4. Table Technique

Table technique represents traceability relations in two dimensions and can represent types of traceability relations. Some work such as (Egyed 2001, Kim et al. 2005, Kotonya and Sommerville 1998, Lindvall and K. 1996, Lindvall and Sandahl 1998, Sommerville and Sawyer 1997) represent traceability relations by applying table technique. The technique is suited for representing one-to-one, one-to-many, and many-to-many relations. Table technique can support representing traceability relations with their semantics.

	MP-UI-1.1	MP-UI-1.2	MP-UI-2	MP-UI-3.1	MP-UI-3.2
MP-UI-1.1			**R**	**R**	
MP-UI-1.2					
MP-UI-2				**C**	
MP-UI-3.1					

Figure 2- 4: Representing traceability by table

Figure 2-4 shows a requirement *MP-UI-1.1* requires the requirements *MP-UI-2* and *MP-UI-3.1* and a requirement *MP-UI-2* has constraints to the requirement *MP-UI-*

3.1. However, it is difficult to maintain with a large number of traceability relations and possible to misread directions of traceability relations.

2.6.5. Mapping Graph Technique

Mapping graph technique represents traceability relations with a graph. A graph shows a scenario which and how software artefacts associate with each other. Mapping graph technique is used to represent with semantics and extra information of traceability relations. It can represent relationships between particular elements in a particular document in a particular file and supports representing of one-to-one, one-to-many and many-to-many relationships. This technique requires to be applied in an automatic way. Examples of traceability approaches using this technique are (Letelier 2002, Maletic et al., Pinheiro and Goguen 1996, Ramesh and Jarke 2001). In (Letelier 2002), a graph is composed of elements of software artefacts, types of traceability relations, and directions of traceability relations. In (Maletic et al.), the authors defined a cluster as a set of documents, while the cluster can be documented in different files. A graph is used to illustrate traceability relations between source code in different clusters and files. Ramesh and Jarke applied KBSA-ADM tool that provides the representation of traceability relations in a graph representing associated artefacts and types of traceability relations.

2.6.6. Mark-up Technique

Mark-up technique represents traceability relations in mark-up languages. Representation of traceability relations with the mark-up technique can be done manually or automatically. The technique supports representing of one-to-one, one-to-many, and many-to-many relationships as well as their semantic. In (Gotel and Finkelstein 1995), they defined a descriptive markup language for recording traceability relations. The descriptive markup language is extended from HTML. The traceability information captured in the markup language includes type of traceability relation, and related software artefacts or stakeholders. The work in

(Spanoudakis et al. 2004, Zisman et al. 2002b) is based on XML technologies. Traceability relations and rules are recorded in XML.

2.6.7. Hyperlink Technique

Hyperlink technique represents traceability relations as cross-links between software artefacts. Hyperlink technique supports representing of extra information and semantics. The technique can represent one-to-one, one-to-many, and many-to-many relationships. The technique allows multiple data-views, complex-interface and interrelated scenarios. The growth of Internet technologies e.g. HTML, XML support the deployment of hyperlink technique. Recent years, the concept of representing traceability relations with the hyperlink technique has been increasingly implemented. Examples are the work in (Knethen 2002a, Ramesh and Jarke 2001, Sherba et al. 2003a, SLATE).

Table 2-2 shows the comparison between different techniques for representing traceability relations.

Table 2- 2: Comparison of techniques for traceability representation

	Identifier	Tagging	Indexing	Table	Mapping graph	Mark-up	Hyperlink
Support representation of traceability relations manually	X	X	X	X		X	
Support representation of traceability relations automatically	X	X	X	X	X	X	X
Support representing of one-to-one relationship	X	X	X	X	X	X	X
Support representing of one-to-many relationship	X	X	X	X	X	X	X
Support representing of many-to-many relationship		X		X	X	X	X
Support representing of semantics		X		X	X	X	X

Existing techniques applied for recording traceability relations are database and special repositories. Some tools (DOORS, Pinheiro and Goguen 1996, Ramesh and Jarke 2001, RTM) represent and record traceability relations by using database. In addition to (Pohl 1996b, Sherba et al. 2003a), the approaches defined system development environments to particularly support traceability activities. The authors defined special repositories for the activities. In (Pohl 1996b), they provided the *process repository* for recording software artefacts and traceability relations according to four types of reference models (as described in Section 2.4). The process repository consists of three levels: (i) *definition schema level*, which the languages for defining processes and traceability reference models are defined; (ii) *definition level*, which the process models and traceability reference models are defined; and (iii) IRD level, which process execution, software artefacts and traceability relations are recorded. In (Sherba et al. 2003a), the authors applied *information integration*

environment (Anderson et al. 2002) in traceability activities. Heterogeneous software artefacts which are translated and integrated into the environment and traceability relations are recorded under the repository in the environment. In (Cleland-Huang et al. 2002a, Cleland-Huang et al. 2002b), traceability relations are recorded in an *event-based traceability* server.

2.7. Traceability Commercial Tools

In this section we summarise the features of commercial tools that support traceability activities. We categorise the tools as: *general-purpose* tools, which are initially developed for supporting other purposes but can be used to support traceability activities; and *specific-purpose* tools, which are developed for supporting requirement management and provide some functionalities for supporting traceability activities.

2.7.1. General-Purpose Tools

Some general-purpose tools are used to support traceability activities. It is simple and practical for small-scale and short-term projects. General-purpose tools, for example, include spreadsheet programs such as Lotus 1-2-3 ™ (Lotus Development Corporation), MS Excel ™ (Microsoft Corporation), and Quattro Pro ™ (Corel Corporation), word processors and hypertext editors such as MS Word ™ (Microsoft Corporation), WordPerfect ™ (Corel Corporation), and Frame Maker ™ (Adobe Systems Inc.). Word processors provide ways to document traceability by using techniques as lists and tables, Hypertext editors can be used to create links between artefacts. Spreadsheet programs help keeping track of different levels of requirements and their attributes.

However, there are three main limitations in using general-purpose tools to capture traceability relations. These limitations are concerned with the facts that (i) configuring of tools is time consuming, (ii) tools can not integrate with other tools nor support many simultaneous users, and (iii) tools do not provide a common and

consistent framework for traceability but promote immediate and ad hoc solutions. Therefore, general-purpose tools are not appropriate for supporting extensive requirements traceability.

2.7.2. Specific-Purpose Tools of Requirements Management

Special-purpose tools provide features such as: (a) creating documents and recording extra information as attributes of documents e.g. date, time, name of creator, version; (b) filtering and sorting to view documents; (c) importing and exporting documents between different projects; (d) maintaining of document versions; and (e) creating reports and summaries. Additionally, some other features may be provided by the tools: (a) graphical user interface; (b) compatibility with other tools; and (c) support for simultaneous users. Examples of some requirements management tools that support traceability include RequisitePro ™ (Rational Software Corporation), Caliber-RM ™ (Technology Builders, Inc.), DOORS ™ (Telelogic AB), RTM Workshop ™ (Integrated Chipware, Inc), and CRADLE ™ (Structured Software Systems Limited) (3SL, CaliberRM, DOORS, RequisitePro, RTM).

DOORS

DOORS (DOORS) is part of a commercial suite of requirements management tools that are produced by Telelogic. It is designed to manage large sets of requirements and handle hundreds of users. DOORS also supports concurrent and remote access by many users at once. The main features of DOORS are to specify software artefacts as well as create and maintain traceability relations between the software artefacts. DOORS claimed that the tool can support tracing and impact analysis of software artefacts by using *impact analysis* and *traceability analysis* relations. The traceability relations are updated when a user has confirmed a proposed change. There is also a function of keeping history logs to record transactions occurred to software artefacts. However, as mentioned before, the generation of traceability relation relies on traceability users and DOORS requires software artefacts to be created by the tool.

RequisitePro

RequisitePro (RequisitePro) is a requirement management tool that is produced by IBM. The tool is designed to improve the communication between different projects and enhance collaborative development. It is integrated with Microsoft word and database to support requirements specification. The tool has functionalities for manually generating traceability relations between requirements created by the tool. The traceability relations supported by the tool are categorized as *evolution* relations and represent the history of changes on the requirements.

CaliberRM

CaliberRM (CaliberRM) is a requirements management tool that is produced by Borland. It is designed to facilitate collaboration, impact analysis, and communication in system development environment. The tool is also aimed to assist the definition and management of a proposed change. However, users are required to identify traceability relations between requirements created by the tool.

As mentioned before, there are limitations in using general-purpose tools for supporting traceability activities and specific-purpose tools such as DOORS, RequisitePro, Caliber-RM, RTM are applied for traceability activities i.e. generation, representation, and usage. However, manual effort is still required to perform the activities. There have been efforts to develop specific tools for supporting requirements engineering and software traceability as described in (Finkelstein 1991, Finkelstein and Fuks 1989, Gotel and Finkelstein 1994, Jones et al. 1995, McMullen 1996-1997). According to the literature, a number of research tools such as (Antoniol et al. 2002, Cleland-Huang et al. 2004, Marcus and Meletic 2003, Pinheiro and Goguen 1996, Pohl 1996a, Sherba et al. 2003a, Zisman et al. 2002b) for software traceability have been developed and integrated into software development environments. However, the following are still considered: (a) some of these tools maintain software traceability for a small project. When the project grows in size, the maintenance of the traceability relations can grow exponentially. This leads the management of software artefacts much more difficult, time-consuming, and error-prone; (b) those tools do not provide for supporting specific-domains of systems

such as product family systems; and (c) some of those tools require manual efforts and have constraints e.g. documents must be specified in pre-defined formats by the tools, or documents must be recorded in special repositories or defined-development-environment.

2.8. Summary

This chapter have provided background information for software traceability. It described the definition, current problems, existing approaches, existing techniques and tools regarding software traceability. In next chapter we provide the review of product family systems.

Chapter 3

Product Family Systems

This chapter describes a literature of product family systems including current problems, and existing approaches, techniques and tools in the domain of product family systems. It also presents the existing approaches and problems of traceability activities in product family systems as well. The motivation and related terminologies are given in Section 3.1. In Section 3.2, we describe current problems in the domain of product family systems. Section 3.3 presents the activities during product family system development. Section 3.4 and Section 3.5 illustrate existing methodologies, approaches, and techniques for product family system development. In Section 3.6, we describe existing tools which are used to support product family system development. Section 3.7 describes the review of traceability of product family systems.

3.1. Introduction to Product Family

Software reuse is the process of software development by using existing software artefacts (Department_of_Defense 1996). Over the last years, approaches and techniques for software reuse have been developed and extended. According to (Clements and Northrop 2002, ESAPS, Weiss and Lai 1999), software reuse at the largest level of granularity is supported by *product family*. This is to serve the reuse practice in an organization having a large number of products, which drives issues such as highly expensive, complex, and tedious tasks. The different exact definition of product family will be given in Section 3.1.1.

The idea of product family was motivated by the need to systematize a number of products more effectively and the fact that these products have a certain set of common and special functionalities. For example, a mobile-phone company has

created a mobile-phone family that contains a set of mobile-phones. Some lower-end mobile-phones have similar basic functionalities but different hardware capacities to offer competitive price. Mobile-phone network communications in some countries provide different standards of transmission and signaling and depend on regional diversity; thereby, a company provides different support for different regions.

3.1.1. Terminologies in Product Family

We describe below terminologies used in the domain of product family system development.

Product Family

Initially, Parnas (Parnas 1976) defined *program family* as a set of software programs constituted as a family whereby a program is developed by applying common properties of prior programs and adding extra properties to the program.

In (Bass et al. 2003, CAFE 2003, Clements and Northrop 2004, Staudenmayer and Perry 1996, Weiss and Lai 1999), *product family* is defined as a set of products sharing some common aspects and having some different aspects. The product family is aimed at gaining the market share under the same business domain and marketing factors. They also suggested *product members* that are products which are built-up by applying shared assets i.e. requirements, architecture, models, and source code in a product family. *Product line* and *business unit* are other terms found in the literature that have the same meaning as that of product family (Ardis and Weiss 1997, Bass et al. 2003, Clements and Northrop 2004).

According to (Bass et al. 2003, CAFE 2003, Clements and Northrop 2002, Staudenmayer and Perry 1996), product family takes into account both hardware and software systems. In (Clements and Northrop 2002), they suggested that a *software product line* is a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or

mission and is developed from a common set of core assets in a prescribed way. In this thesis, we focus on and call the software systems that are developed for product family as *product family systems.*

Features

The term *feature* has been initially used in (Kang et al. 1990). The authors defined a feature as a prominent and distinctive aspect or characteristic of a system that is visible to various stakeholders (e.g. end-users, domain experts, developers). In (Bosch 2000, Gibson et al. 1997, Griss 2000, Svahnberg et al. 2001), a feature is concerned with a logical behavior of a system that is specified as a requirement or set of requirements (i.e. functional and non-functional requirements). In (Bailin 1990), the author suggested a different definition whereby a feature refers to any distinctive or unusual aspect of a system that requires a decision for system engineering. In this thesis, we use the term feature as a user-visible aspect or as a characteristic of product family systems. A feature is related to other features and represented in a tree structure of And/Or nodes to express common and variable aspects within product family systems

Core Assets

Core assets (Clements and Northrop 2004) are those assets that form the basis for a product family. Core assets include requirements, architecture, and reusable software components, domain models, documentation and specifications, schedules, test cases, and work plans. In (Riebisch et al. 2002), the authors also suggested that core assets i.e. requirements, architectures, analysis models, design models, test cases, and source codes are reused between different product members in a product family. A variant term of core asset is *platform* that is defined in the domain of Model-Driven Architecture (MDA).

Commonality Vs. Variability

According to (Bosch 1998, Clements and Northrop 2004, Weiss and Lai 1999), *commonality* is concerned with a set of similar functionalities or aspects between

product members of a product family and *variability* is defined as different functionalities or aspects between product members of a product family.

3.2. Problems of the Establishment and Maintenance of Product Family Systems

Many approaches have been proposed to support the development of product family systems. However, there are many associated problems which we describe in this section

I. The Difficulty to Get Support from Organisations

Due to timing constraints, an organisation usually considers available methodologies rather than establishing product family. Additionally, an organisation has defined and used the current development process for a certain period of time. The organization prefers adopting familiar and practical techniques to support the development process rather than unfamiliar techniques.

II. The Uncontrolled Growth of Variety

Ideally, the establishment of product family needs to have a stable and clear vision of domain; however, it needs to be flexible enough to evolve new requirements. Practically, an organization is uncertain about requirements of product members and develops extra options to anticipate all possible requirements (Bosch 2001, Sinnema 2004, Svahnberg and Bosch 2000, Thiel and Hein 2002).

III. The Difficulty in Communication

Product family system development is a collaborative process where people from various disciplines need to communicate each other. In other words, communication is required to facilitate and improve the software system development. For example, Meyer (Meyer 1998) suggested that the interaction between stakeholders e.g. between the development team and manufacturing team should be concerned. In addition to (Finkelstein and Guertin 1998), the authors proposed that good communication provides the right requirements at the right

time and the right place. Precise requirements must be known in order to facilitate actual implementation.

However, it is not easy to support communication between various groups of stakeholders in an organisation. Successful communication between stakeholders depends on various factors such as: (i) sufficient resources e.g. staff or tool to facilitate the communication; (ii) differences in organisational cultures; (iii) distinct organisational structures; and (iv) stakeholders' attitudes and aspirations. Unsuccessful communication in an organization leads to misunderstanding and lacks of some concepts during the development of software systems.

IV. The Difficulty of Defining Commonality and Variability

Defining commonality and variability of product family is to thoroughly discover the product family descriptions including all common and possible variable aspects. However, there are two issues which cause the difficulty of the practice (Halmans and Pohl 2003, Svahnberg and Bosch 2000, Webber and Gomaa 2002, Weiss 1998). These issues are:

Different Perspectives

It is difficult to share views between different products and represent opinions between different tools. For example, sales engineers can offer a new combination of requirements, which seem perfectly reasonable from a customer viewpoint, but appear to be unproved in the technology domain. This difficulty to describe different perspectives of an artefact causes the difficulty of defining commonality and variability.

Lack of Knowledge

Defining commonality and variability of a product family needs stakeholders who have enough experience, knowledge, responsibility and authority. However, it is not easy to find stakeholders who are qualified and also available to take this task in charge (Bosch 1998).

V. The Difficulty of Documenting Management

Data in product family systems rapidly grow as the number of product members in a product family increases. Bosch (Bosch 1998) described that stakeholders need to interpret documents and discover relevant documents; therefore, it is important to specify the documents clearly and validly. However, there is a large number and heterogeneity of artefacts and relationships between those artefacts in the domain of product family systems. It is difficult to document the semantics between documents. The difficulty of documenting management leads the following issues: (i) *missing semantics* – documents miss to express the semantics of the context; (ii) *failure of interpreting the semantics* – stakeholders fail to interpret the semantics of documents; (iii) *missing of relevant documents* – stakeholders miss discovering all related documents of interest to them; and (iv) *failure of searching documents* – it is difficult to locate the documents efficiently and promptly.

VI. The Confliction and Dependency between Artefacts in Product Family Systems

Ideally, a feature is an atomic unit and a set of features can be put together to fit with a product member's requirements. However, features are not actually independent. Adding or removing a feature to or from a product family has an impact on other features. Additionally, a feature is also related to other types of artefacts in a product family. Therefore, adding or removing an artefact has also an impact on other different artefacts. It leads a difficulty to development and maintenance of product family systems.

VII. The Difficulty to Specialise Variability

Variability can be specialized in different phases i.e. design, implementation, compile, linking, or run-time. However, there are some difficulties in specialization for variability such as: (i) *feature interaction* – specialization of a feature can lead other features in a product family to have unexpected results (Bosch 2000).; and (ii) *separation of concern* – some variability are separated into different artefacts; however,

this can lead to the difficulty of specialization (Gomaa and Shin 2004, Svahnberg et al. 2001).

VIII. Issues of Evolution of Product Family Systems

There are some situations that require the evolution of product family systems such as: (i) there is a change on existing product family; and (ii) the core assets of a product family have missed some functionalities. These situations occur when the maturity level of product family systems in an organization has grown. The organisation requires a software process which implements new requirements and maintains the consistency of existing systems. However, the issues of evolution are found and defined in (Bosch 2000).

3.3. Activities in the Process of Product Family System Development

According to the maturity level of an organization, the approaches for the development of product family can be categorised, namely *proactive*, *reactive*, and *extractive*. We describe below three types of approaches for the product family system development.

Proactive

The *proactive* approach (Krueger 2001) is an approach of the product family system development when an organization decides to analyse, design, and implement a line of products prior to the creation of individual product members. The product family is built-up and the core assets representing the commonality and variability are created. All product members are then created under the scope of the product family. The approach is viewed as a top-down developing strategy which requires the setting of broad goals and the goals are refined in later phases of the development.

Reactive

The *reactive* approach (Krueger 2001) is an approach of the product family system development when an organization enlarges the product family systems in an

incremental way based on the demand for new product members or new requirements for existing products. The core assets need to be extended and evolved in such a way as to correspond to new requirements or new systems. This is caused by the fact that the customer requirements considerably influence the architecture and the design of products. On the other hand, a company that sticks strictly to the principles of made-to-order manufacturing will not allow an uncontrolled proliferation of variety due to the demands of individual customers. However, in reality, many companies have a production control concept based on customer requirements. So the problem occurs when the architecture and design of product family systems should be maintained. This level of development takes shorter time than the previous one since system developers only extend and adapt the available products.

Extractive

The *extractive* approach (Krueger 2001) is an approach of the product family system development when an organisation creates product family systems based on existing product members by identifying and using common and variable aspects of these products. The stakeholders i.e. domain experts and system developers analyse and define the product family by taking into consideration individual products' requirements. The approach is viewed as a bottom-up developing strategy that begins with existing artefacts e.g. requirements specification, design and source code, then creates the higher granularity level of each artefact as the core assets.

In the following section, we describe the activities occurred during the product family system development process. In addition to (Bosch 2000, Clements and Northrop 2002, Jazayeri et al. 2000, Thiel and Hein 2002), *software product line engineering* is a methodology for developing product family systems that focuses on activities of analysis, design, and implementation of a product family as well as the use of the core assets inclusive common and variable artefacts potentially and effectively for product members.

Figure 3-1 illustrates the main activities of software product line engineering i.e. domain engineering and application engineering.

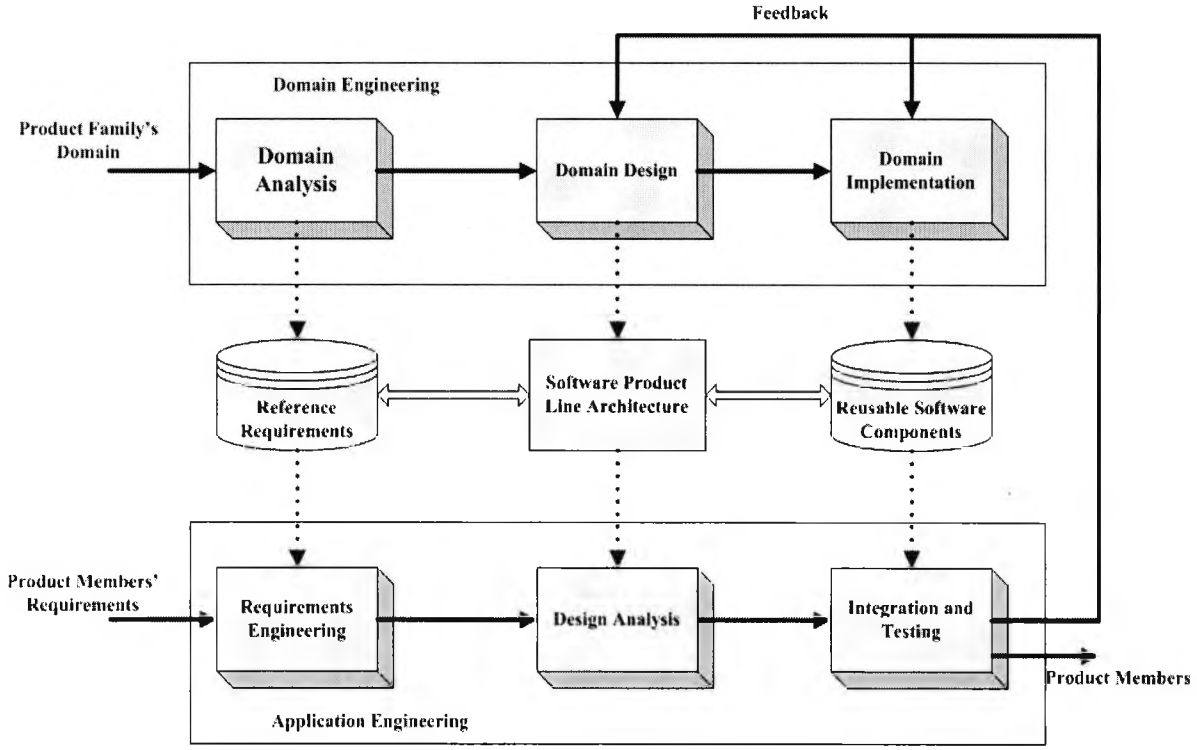


Figure 3- 1: Activities in software product line engineering adopted from (Clements and Northrop 2004)

3.3.1. Domain Engineering

Domain engineering is a systematic process for the creation of the core assets (Clements and Northrop 2004). There are three steps for domain engineering:

Domain Analysis

> Domain Analysis is the process of identifying, collecting, organizing and representing the relevant information in a domain, based upon the study of existing systems and their developing histories, knowledge captured from domain experts, underlying theory, and emerging technology within a domain (Kang et al. 1990).

As shown in Figure 3-1, software artefacts that are produced during the activity of domain analysis are called *reference requirements*. The reference requirements define the products and their requirements in a family. The reference requirements contain commonality and variability of the product family. The following sub-activities occur during the domain analysis:

I. Scoping

According to (Arango and Prieto-Diaz 1991, Ardis and Weiss 1997), domain analysis for a product family basically starts from *scoping*. Scoping is to identify the context of product members in a product family e.g. functionalities and performances. The activity is concerned with domain knowledge obtained from domain experts and other sources such as books, user manuals, and design documents (Nuseibeh and Easterbrook 2000). The domain experts analyse and define the boundary of the product family and the standard terminologies in the family. The product members are therefore defined.

II. Commonality and Variability

The activity of defining commonality and variability is to thoroughly discover and define commonality and variability in a product family (Ardis and Weiss 1997, Weiss 1995). Many existing approaches are proposed to support the activity. Examples of such approaches are (Ardis and Weiss 1997, Bosch 2000, Clements and Northrop 2002, Svahnberg and Bosch 2000, Weiss 1995). The determination of whether a characteristic is a commonality or variability mostly depends on a strategic decision of organisations.

In particular, defining commonality is the determination of whether a requirement is served as the commonality of a product family. Defining variability is the determination of whether a requirement is served as the variability of a product family. Variability is represented as a set of *variation points*. Each variation point is a situation that product members can be specialized differently and dependent on a number of *variants*. Variants are possible variables for each variation point. A

variation point is classified as: (i) *optional* – an aspect may exist in a product; (ii) *alternative* – an aspect can be specialized as one of the variants; and (iii) *optional alternative* – an aspect can be specialized as one of the variants or does not exist (Svahnberg et al. 2001). Variation points can appear at different phases of product family system development i.e. analysis, design, and implementation. At the state of domain analysis, a variability point is concerned with the highest abstraction level of an artefact.

III. Planning for Product Members and Features

According to (Arango and Prieto-Diaz 1991), one of the activities in domain analysis is to identify features of product members in a product family. The features of a product family are planned for possible product members. In other words, the relevant requirements of product members are associated to the features of a product family. The common and variable aspects of a product family are accommodated and planned for product members.

Domain Design

> Domain design is the process of developing a design model from the products of domain analysis and the knowledge gained from the study of software requirements or design reuse and generic architectures (Garlan and Shaw 1993).

Software artefacts that are produced during the activity of domain design are called *software product line architecture* (see Figure 3-1). In (Bass et al. 2003, Jazayeri et al. 2000), *software architecture* forms the backbone of integrating software systems and consists of a set of decisions and interfaces which connect software components together. Software product line architecture differs from an architecture of single systems that it must represent the common design for all product members and variable design for specific product members (Linden et al. 2004). The following sub-activities occur during the domain design:

I. Software Product Line Architecture Definition

The activity of software product line architecture definition is to design the software architecture that describes commonality and variability of product members. The software product line architecture is composed of a set of architectural decisions, a set of reusable design artefacts, and a set of optional design artefacts.

The variability in software product line architecture is called *designed variability points* (Svahnberg et al. 2001). The software product line architecture can be elaborated into different levels of granularity. At higher levels, the software product line architecture does not entail shared artefacts between product members while at the low levels, the software product line architecture make a distinction between specific designs of product members.

II. Software Product Line Architecture Evaluation

The activity of software product line architecture evaluation is to evaluate the software architecture that describes commonality and variability of product members. The evaluation of software product line architecture is to assure that the architecture has the right properties and characteristics of a product family.

For the evaluation of software product line architecture, the following must be considered: (i) the context for software product line architecture must be scoped and planned during domain analysis; (ii) the commonality of a product family must be elaborated in several levels of the architecture; and (iii) the variability of a product family must be identified and provided with a set of variants for each designed variability point in the software product line architecture.

However, Bosch (Bosch 2000) suggested that the maturity of software product line architecture can be viewed as three levels: (i) an *under-specified* architecture that defines common aspects but does not specify differences between product members; (ii) a *specified* architecture that defines both common and variable aspects for product members; however, does not define possible variables for variable

aspects; and (iii) an *enforced* architecture that defines both common and variable aspects covering possible variables for all product members.

Many approaches and techniques are proposed to support domain design for a product family. Relevant existing methodologies e.g. *model-based software engineering* (MBSE 1993), *organizational domain modeling (ODM)* (Simos 1995), *synthesis* (Campbell et al. 1990), *domain-specific software architecture (DSSA) program* (Tracz et al. 1993), *evolutionary domain life-cycle (EDLC)* (Gomaa et al. 1989) are applied for the development of software product line architecture. Some general-purpose techniques such as data flow diagrams, structured analysis and design techniques, entity relationship modeling (ERM), object models (e.g. UML (UML)), view point-oriented models (Finkelstein et al. 1990) can be also applied for the activity. Recently a number of methodologies such as (Atkinson et al. 2000, Batory et al. 2000, Bayer et al. 1999, Griss et al. 1998, Kang et al. 1998, QADA, Simos 1995, Weiss 1995, Weiss and Lai 1999) are proposed to particularly support the activity of domain design in the domain of product family.

Domain Implementation

> Domain implementation is the process of identifying reusable components based on the domain model and generic architecture (Clements and Northrop 2004).

Software artefacts that are produced during the activity of domain implementation are called *reusable software components* (see Figure 3-1). The activity is focused on the creation of reusable software components e.g. source codes and linking libraries that are later assembled for product members. In (Szyperski 1997), a reusable software component is a unit of composition with interfaces and independent context. The reusable software component is created and then integrated with other reusable software components for a particular product member. The set of reusable components are defined independently and provide the connectors for integration

with other components to fit into a specific functionality. The components are viewed as black boxes whose data and implementation details are completely hidden and only interfaces are allowed. The development of components can be applied with relevant existing methods such as object-oriented methods e.g. (Bosch 2000, Szyperski 1997).

At the end of the domain engineering process, an organization is ready for developing product members. In the following section, we describe the activities for developing the product members in the software product line engineering.

3.3.2. Application Engineering

As shown in Figure 3-1, application engineering is another major activity of software product line engineering. According to (Northrop 2002), *application engineering* is a systematic process for the creation of a product member from the core assets created during the domain engineering. Domain engineering assures that the activities of analysis, design and implementation of a product family are thoroughly performed for all product members, while application engineering assures the reuse of the core assets of the product family for the creation of product members.

The application engineering process for a product family is comparably considered with the process for a single system (Clements and Northrop 2004). There are activities such as: (i) *requirements engineering,* which is a process that consists of requirements elicitation, analysis, specification, verification, and management (Fairley and Thayer 1997, Sommerville and Sawyer 1997, Sutcliffe and Maiden 1998); (ii) *design analysis,* which is a process that is concerned with how the system functionality is to be provided by the different components of the system (Sommerville 2000)*;* and (iii) *integration and testing,* which is a process of taking reusable components then putting them together to build a complete system, and of testing if the system is working appropriately.

Requirements Engineering

The activity of requirements engineering focuses on identifying, colleting, organizing and representing requirements of a product member. The major difference between requirements engineering of an individual product and a product member is that stakeholders not only focus on the specific product but also on the scope of product family. Technically, the requirements of product members are defined and scoped under the domain of the product family's requirements. A variability point of a requirement is bound with a variant for a particular product member during requirements engineering.

Design Analysis

Design analysis in application engineering must be consistent with the concept of design analysis in domain engineering. This activity is to analyse and design the architecture for a product member. Software product line architecture is refined and specialized for a particular product member. The software architecture of the product family is configured to fit for a product member based on the specific product's requirements. The configuration includes the addition and removal of designed variability points of the product family.

In (Bosch 2000), *architecture pruning* is an activity that the common aspects of software product line architecture is collected and the variable aspects for a specific product member are specified. The composition of common and variable aspects acquires the software architecture for a specific product member. Nonetheless, it is possible that a software product line architecture does not fulfill the complete design of a specific product. This needs an activity called *architecture extension* (Bosch 2000). The activity extends some aspects that are not included in the software product line architecture.

Integration and Testing

The usage of the core assets of product family and development of product members involve the following three steps: (i) discovering a set of reusable

components for a specific product member; (ii) instantiating the variability points of the reusable components for a specific product member; and (iii) integrating and testing the reusable components for the product member.

3.4. Methodologies for the Development of Product Family Systems

In this section, we describe existing methodologies to support product family system development. These methodologies can be classified into two groups as (a) object-oriented and (b) feature-oriented methodologies. We describe below these approaches.

3.4.1. Object-Oriented Methodologies

Object-oriented methodologies have been common and popular in the development of software systems. Many existing object-oriented methods are aimed at supporting the development of single software systems. Recently, some object-oriented methods have been extended and proposed for the development of product family systems. We describe below the methods and approaches for product family system development in the object-oriented paradigm.

COPA

Component-Oriented Platform Method (COPA) (America et al. 2000) is proposed for product families of software-intensive electronic products i.e. telecommunication, medical imaging, and consumer electronics. COPA defined *architectural* and *process frameworks*. The architectural framework consists of five views:

(i) *Customer view* – the view shows customer business models represented in customers language or textual language.

(ii) *Application view* – the view shows application models represented in UML diagrams

(iii) *Functional view* – the view shows functionalities and performances of systems represented in use cases

(iv) *Conceptual view* – the view presents *platform* and *product-specific* components created for a product family and product member, respectively. In COPA, construction components are applied with some component-based techniques such as COTS, Microsoft's COM component model, Sun's JavaBeans, and OMG's CORBA.

(v) *Realization view* – the view illustrates specific techniques e.g. hardware infrastructure, hardware platform, operating systems. These are specified in a textual language.

The process framework consists of three main activities:

(i) *Product family engineering* – this activity is driven by policy and plans of an organisation. There are sub-activities during product family engineering such as domain modeling, requirements formulation, and commercial and technical design. These activities construct customer, application, and functional views. The architecture of product family is created during product family engineering. For example, COPA applied Koala for representing the product family architecture. According to Figure 3-1, this activity can be comparable with the domain analysis and domain design during domain engineering.

(ii) *Platform engineering* – this activity is concerned with technology and people management. Sub-activities can occur during platform engineering such as standard development, cooperating between stakeholders in product family engineering and product engineering to comprehend requirements of product family and product members, integrating and testing for product members, and maintenance of existing reusable components and platforms. This activity has sub-activities that are comparable with domain engineering including domain analysis, domain design, and domain implementation as shown in Figure 3-1.

(iii) *Product engineering* – this activity is concerned with the customer-oriented process. There are sub-activities during product engineering such as standard development, cooperating with customers to understand specific requirements, constructions of product members, and maintenance and support for product members. According to Figure 3-1, this activity can compare with application

engineering including requirements engineering, design analysis, and integration and testing.

In the COPA method, the authors suggested the activities in software product line engineering and artefacts created during three activities. The artefacts are represented in UML diagrams, use cases, textual language, Koala language (Ommering et al. 2000), and component-based representation languages.

QADA

Quality-driven Development of Software Family Architectures QADA (QADA) is a quality-driven architecture-centric method for product family system development. The QADA method described the development of software product line architecture. The method includes five activities:

(i) *Requirements engineering* – this activity is aimed to capture and analyse requirements and context model. The requirements i.e. functional and non-functional requirements and context model i.e. hardware and software interfaces of a system, a set of constraints, rules, and standards are represented in textual language.

(ii) *Conceptual architecture design* – this activity is aimed to identify a conceptual architecture which is represented with three views namely, *structural view, behavior view*, and *deployment view*. The structural view is concerned with conceptual components and their relationships. The structural view is composed of three types of artefacts: (a) list of functional responsibilities represented in textual language; (b) table of non-functional requirements represented in text and table; and (c) decomposition model. The behavior view is concerned with dynamic actions and kinds of actions to which a system produces. The behavior is represented in a collaboration model. The deployment view is concerned with allocation of the conceptual components into hardware components. The behavior is composed of two types of artefacts: (a) table of deployment units represented in text and table; and (b) allocation model. Another type of artefact

generated during conceptual architecture design is *design rationale* which represents design principles and rules.

(iii) *Conceptual architecture analysis* – this activity focuses on qualities, commonality, and variability of a system. Three types of artefacts are created namely: (a) *product line scope*, which represents a boundary of product family; (b) *taxonomy of requirements*, which describe syntactic architectural notations and are represented in domain models, relevant architectural views; architectural styles; environmental assumptions and constraints; and trade-off rationale; and (c) *knowledge base*, which allows the evaluation of collections of architectural styles and patterns in terms of both quality factors and concerns. The knowledge base in QADA contains materials, quality attributes, questions that describe the evaluation of artefacts.

(iv) *Concrete architecture design* – this activity focuses on providing a set of concrete software components and definition of interfaces between components. The activity is concerned with three views in the activity of concrete architecture design (structural view, behavior view, and deployment view). Firstly, the list of functional, non-functional requirements, and decomposition model from the conceptual architecture is designed and refined as *structural diagrams* that represent concrete components, interfaces and relationship. Secondly, the collaborative model from the conceptual architecture is defined and refined as *state diagrams* and *message sequence charts.* Thirdly, the table of deployment units and allocation model from the conceptual architecture are designed and refined as *deployment model.*

(v) *Concrete architecture analysis* – this activity is aimed to assess and evaluate the software product line architecture regarding expected changes. The analysis method consists of five sub-activities: (a) deriving of changes from the product line scope; (b) defining product-line architecture description; (c) defining scenario identification; (d) evaluating the effect of scenarios; and (e) identifying scenario interaction.

In the QADA method, the activities of domain engineering are defined. More specifically, the activity of requirements engineering is comparable with domain

analysis in Figure 3-1, and the activities of conceptual architecture design, conceptual architecture analysis, concrete architecture design, and concrete architecture analysis are comparable with domain design in Figure 3-1. However, the QADA method does not cover an activity of application engineering in Figure 3-1. In addition, artefacts created during theses activities are represented by using textual language and UML diagrams.

KobrA

KobrA (Atkinson et al. 2000) is a component-based method for software product-line engineering that is developed by Fraunhofer IESE. In the KobrA method, the authors proposed a *Komponent* as a set of reusable components that satisfy a requirement or group of requirements. The Kobra method is divided in two main activities: (i) *framework engineering*, which defines a set of Komponents; and (ii) *application engineering*, which applies existing Komponents and constructs a product member.

Framework engineering consists of four activities, namely:

(i) *Context realization* – the aim of this activity is to define properties and scope of a product family. The *business process models*, which describe the requirements and constrains of a product family, and *decision models*, which describe common and variable requirements of a product family, are created.

(ii) *Komponent specification* – the aim of the activity is to describe properties of a Komponent. The *structural model*, which is represented in UML class diagrams, *behavioural model*, which is represented in UML statechart diagrams, *functional model*, which is represented in Operation schemas, and *decision model*, which is represented in a textual language, are created.

(iii) *Komponent realisation* – the aim of the activity is to define the design of a Komponent. The *interaction model*, which is represented in UML collaboration diagrams, *structural model*, which is represented in UML class diagrams, *activity model*, which is represented in UML activity diagrams, and *decision model*, which is represented in a textual language, are created.

(iv) *Component reuse* – this activity focuses on applying existing components to develop new Komponent.

Application engineering consists of two activities:

(i) *Context realization instantiation* – the activity is aimed to identify relevant Komponents to be reused for a product member.

(ii) *Framework instantiation* – the activity is used to create a framework of a set of Komponents and relationships between those Komponents for a product member.

The Kobra method is defined to complete the activities in the development of product family systems. More specifically, the activities of context realization, Komponent specification, Komponent realization, and component reuse are comparable with the activities of domain analysis, domain design, and domain implementation as shown in Figure 3-1, respectively. Moreover, the activities of context realization instantiation, and framework instantiation cover the activities of application engineering including requirements engineering, design analysis, and integration and testing in Figure 3-1. Additionally, the method is systematic, scalable and practical for the development of product family systems. The artefacts created in the method are based on UML diagrams and textual language that are customised to fulfil the activities in the domain of product family systems.

PuLSE

Product Line Software Engineering (PuLSE) (Bayer et al. 1999) is a customizable software product line engineering approach. The PuLSE method consists of four main activities:

(i) *Initialisation* – the activity is aimed to analyse and evaluate a situation of an organisation.

(ii) *Infrastructure construction* – the aim of this activity is to define a scope and processes of a product family. A scope model and definitions of a product family are created.

(iii) *Infrastructure usage* – the aim of activity is to define and create product members.

(iv) *Evolution and management* – the aim of activity is to evolve the product family.

The PuLSE method consists of six technical components and three support components. The technical components are: (i) PuLSE-BC, which is used to support the analysis and evaluation of an organisation in the initialisation activity; (ii) PuLSE-Eco, which is used to support an economic analysis of a product family; (iii) PuLSE-CDA, which is used to support a domain analysis of a product family; (iv) PuLSE-DSSA, which is used to support a domain design of a product family; (v) PuLSE-I, which is used to support the development of product member; and (vi) PuLSE-EM, which is used to support the evolution and management of product family.

The support components are: (i) *project entry points*, which are used to support analysis of an organisation' situation; (ii) *maturity scale*, which are used to support evaluation the adoption of product family; and (iii) *organization issues*, which are used to support maintenance of product family.

PuLSE defined the framework of components conducted by different activities. The activity of initialization is comparable with domain analysis in Figure 3-1. The activity of infrastructure construction has sub-activities in common with domain analysis, domain design, and domain implementation. Moreover, the activity of infrastructure usage is comparable with application engineering including requirements engineering, design analysis, and integration and testing as shown in Figure 3-1. In addition, software product line architecture and other artefacts in a product family are represented as a set of prescribed components.

FAST

Family-oriented Abstraction, Specification and Translation (FAST) (Weiss 1995) is a software product line method that initially described two main activities in software

product line engineering. The activities, which resemble the main activities depicted in Figure 3-1, are:

(i) *Domain engineering*, which defines a product family and the core assets of the product family; and

(ii) *Application engineering*, which develops product members by using the core assets of the product family.

FAST describes a domain specific language AML (Application Modeling Language) for specifying the requirements of a product family. The requirements of a product family represented in the language are then specialized for product members. However, the definition and specification of requirements are restricted.

RSEB

Reuse-Driven Software Engineering Business (RSEB) (Jacobson et al. 1997) is proposed to focus on achievement of business goals and improvement of business performance. In (Jacobson et al. 1997), they proposed to apply use cases to describe reference requirements of a product family and UML diagrams to describe the software product line architecture. They also defined activities in the development of product family systems:

(i) *Requirements engineering*, where variability is specified as use cases;

(ii) *Architectural family engineering*, where the software product line architecture is created in UML diagrams;

(iii) *Component system engineering*, where reusable components are developed; and

(iv) *Application system engineering*, in which product members are developed.

The activities defined in RSEB are comparable with ones shown in Figure 3-1. More specifically, the activity of requirements engineering in RSEB is concerned about domain analysis and requirement engineering defined in (Clements and Northrop 2004). The activities of architectural family engineering, and component system engineering have likewise sub-activities in domain design and domain implementation, respectively. Moreover, the activity of application engineering in

RSEB covers the activities of design analysis, and integration and testing as shown in Figure 3-1.

SPLIT

Software Product-Line Integrated Technology (SPLIT) (Coriat et al. 2000) is a systemic approach for the development of product family systems. SPLIT suggested a life-cycle of the development process which consists of two activities. The activities, which resemble the main activities depicted in Figure 3-1, are:

(i) *Domain engineering*, which reference requirements, software product line architecture, and reusable components are created; and

(ii) *Application engineering*, which product members are developed.

There are four approaches applied in SPLIT:

(i) The approach called *SPLIT/Cloud* is applied to develop the reference requirements of product family systems. In this activity there are artefacts created: business process, capability, functional area, force, functional requirement, and non-functional requirement. In SPLIT, they described two situations of requirements engineering: the first one is the development based on existing products; and the second one is the development from scratch. The product family system development based on existing products consists of activities: (i) define reference requirements i.e. functional and non-functional; (ii) identify and organize the requirements of each product member; (iii) define artefacts that represent high-level views of functional requirements of each product member; (iv) define artefacts that represent high-level views of non-functional requirements of each product member; (v) map high-level views of functional and non-functional requirements to the reference requirements

The product family system development from scratch consists of activities: (i) define the domain of a product family; (ii) scope the domain; (iii) identify the requirements of the product family; (iv) determine COTS used in the product family domain by applying with COTS model; (v) define reference requirements

i.e. functional and non-functional; (vi) define a business process; (vii) define capabilities related to each business process; and (viii) define forces related to each non-functional aspect.

(ii) The approach called *Daisy* is applied for developing software product line architecture. In Daisy, a software system product line architecture (SSPLA) description is based on three architectural views: (a) business view; (b) subsystem view; and (c) technology view. The business view represents subject area and analysis pattern. The subsystem view represents subsystem, architectural pattern, process, architectural guidelines, architectural constraints and information. The technology view represents component model, computing infrastructure and deployment. The views are represented in UML diagrams.

(iii) The approach *Ladder* is applied for developing reusable components. In Ladder, they suggested the transformations, composition, splitting up, abstraction, refinement, development branch for reusable components development as well as COTS adaptation.

(iv) The approach *Wheels* is applied for supporting sub-processes during domain engineering and application engineering in SPLIT.

The SPLIT method is applied in ESAPS (ESAPS) and CAFÉ (CAFE 2003) projects. The method itself is composed of other methods to support each activity in software product line engineering. Otherwise, artefacts produced by using these methods are represented in i.e. UML diagrams, use cases, component-based representation languages.

3.4.2. Feature-Oriented Methodologies

The concept of feature-orientation is not completely new in software engineering and there have been efforts to apply the concept of features to express aspects of a software system. Examples of feature-oriented methods are FODA (Kang et al.

1990), FORM (Kang et al. 1998), and FeatuRSEB (Griss et al. 1998), which are increasingly important to software product line engineering due to several reasons:

(i) The fact is when developing the product family, stakeholders communicate with each other in terms of product features. It becomes an effective media of communication between customers and system developers.

(ii) Due to a large size and diversion of requirements for product family systems, specifying and representing the requirements becomes primary tasks in domain analysis as these activities are supported by the feature-oriented methodologies.

(iii) Features can be used as the basis for analyzing and representing commonalities and variabilities of product members under the same product family. Additionally, the feature-oriented methodologies offer a way to classify various requirements.

FODA

Feature-Oriented Domain Analysis (FODA) (Kang et al. 1990) is proposed to support the activity of domain analysis. In FODA, the activities are described and cover the activity of domain analysis depicted in Figure 3-1. Three activities are:

(i) *Domain analysis*, which focuses on scoping of a product family and identifying product members;

(ii) *Feature analysis*, which develops a list of common and variable aspects of a product family; and

(iii) *Feature modelling*, which models the common and variable aspects as *a feature model.*

FODA is an initial method that defines a feature model for representing common and variable aspects of a product family. Identification of features requires domain knowledge obtained from the domain experts and other sources such as books, user manuals, design documents, etc. In FODA, the authors described that domain experts and system analysts can use standard terminologies to communicate with each other in mature and stable domains. Therefore, analyzing the domain

terminology is an effective and efficient way to identify the features of a given domain. However, in prior to feature identification, standard terminologies and domain scope should be done since they are not available in immature or emergent domains. Feature models are used as a mechanism to facilitate different perceptions of domain concepts and scope which cause confusion between stakeholders.

The authors defined three types of features: (i) *mandatory* features, which represent the commonality of a product family; (ii) *alternative* features, which are specialized for product members; and (iii) *optional* features, which may or may not exist in product members. The feature model consists of elements such as: (a) a *tree-structured diagram* which represents characteristics of product family; (b) a *definition* for each feature; and (c) *composition rules* which are defined rationally between features. There are two types of rules: (i) one feature *requires* another feature: and (ii) one feature is *included* in another feature.

FORM

Feature-Oriented Reuse Method (FORM) is an extension of FODA that provides the activities of domain analysis and the development of core assets. Three activities are concerned:

(i) *Feature modelling* – that is a process for defining features of product family systems. The authors proposed to apply the extension of the feature model from FODA for representing features. They proposed the classification of with respect features to their purpose as: (a) a set of *capability* features that express the characteristics of distinct services, operations, functions, or performances, (b) a set of *operating environment* features that represent attributes of the environment in which an application is used and operated, (c) a set of *domain technology* features that represent the domain of realization (e.g., navigation methods in the aviation domain), and (d) a set *of implementation technique* features that represent implementation details at lower and more technical levels e.g. abstract data types and sorting algorithms. Kang et al. pointed out that a domain technology feature is more specific to a given domain and may not be usable in other

domains while an implementation technique feature is more generic and may be used in other domains.

(ii) *Architecture modelling* – that is a process for defining software product line architectures. Artefacts created during this process are viewed a hierarchy and consists *subsystem model, process model* and *module model.* These models are represented the commonality and variability of the product family.

(iii) *Component engineering* – that is a process for defining reusable components. In (Lee et al. 2000), the authors described the technique used in the activity of component engineering in the FORM method. The authors described principles for the creation of reusable components by mapping features created during the activity of feature modeling. The principles are (a) capability features can be modeled as an object or group of objects that provide a similar set of operations. The object or group of objects is specified with a parameter for a particular product member; (b) operating environment features can be modeled as an object or group of objects that provide a set of operations for different requirements of product members; (c) domain technology features are modeled to be specific for the domain of product family; and (d) implementation technique features should be used to implement domain-specific objects. For example, a communication method feature (e.g. synchronized or asynchronized communication) depends on the implementation languages or platforms. However, the mapping of the feature model and product member is not described.

In the FORM method, the activities of domain engineering are defined. More specifically, the activities of feature modeling, architecture modeling, and component engineering have likewise sub-activities in domain analysis, domain design, and domain implementation defined in (Clements and Northrop 2004). However, the FORM method does not cover an activity of application engineering.

FeatuRSEB

Featuring RSEB (FeatuRSEB) (Griss et al. 1998) is a combination of RSEB method (Campbell et al. 1990) and FODA (Kang et al. 1990). The FeatuRSEB method includes the activities defined in RSEB which are requirements engineering, architectural family engineering, component system engineering, and application engineering. The method adapted using a feature model by adding UML-based relationships i.e. dependency and refinement. The feature model is used to represent common and variable RSEB models. In other words, the feature model is used to represent an association between RSEB models in a product family.

3.5. Techniques for the Development of Product Family Systems

In this section, we describe existing techniques to support product family system development. These are (a) use cases, (b) UML modeling, (c) feature modeling, and (d) architecture description and component-based languages.

3.5.1. Use Cases

In general, a use case (Cockburn 1997) is a textual specification language that is used for specifying requirements. Examples of the approaches proposed to apply use cases in the activities of product family system development are (America et al. 2000, Griss et al. 1998, Jacobson et al. 1997). In (Fantechi et al. 2004), the authors proposed to express the requirements of product members in a product family by extending the use case definition given by Cockburn (Cockburn 2000). The variability is expressed in use cases by using special tags. The tags indicate the variable requirements of a product family that need to be specialized for a product member. They proposed three types of tags: (i) *alternative tag*, which represents variable requirements with a predefined set of requirement variants; (ii) *parametric tag*, which represents variable requirements that requires the instantiation of specific parameters for a product member, and (iii) *optional tag*, which represents variable requirements which may or may not be instantiated for a product member.

John at el. (John and Muthig 2002) extended use case specifications for representing variable requirements. Use cases express variation points and variants for product members. In addition, the authors applied the decision model to express the relationships and dependencies between the variable requirements.

3.5.2. UML Modeling

According to the literature (America et al. 2000, Atkinson et al. 2000, Coriat et al. 2000, Griss et al. 1998, Jacobson et al. 1997, QADA), object modeling technique is used in software product line engineering. Some approaches such as (Clauss 2001, Gomaa 2004, Keepence and Mannion 1999) are proposed to adapt UML diagrams for modeling software product family systems. Gomaa (Gomaa 2004) proposed *product line UML-based software engineering* (PLUS) by using UML modeling for the development of product family systems. PLUS applied UML diagrams to represent the commonality and variability of product family systems.

In (Clauss 2001), they use a UML class diagram to represent software product line architecture. They define three types of classes for expressing variability in a product family: (i) *variationPoint*, which represents a variation point of a product family; (ii) *variant*, which represents an alternative of a particular variation point; and (iii) *optional*, which represents an optional class . They applied two types of relationships to assist representation of variability: (i) *generalization/specialization*, which associates between classes typed of variationPoint and variant; and (ii) *association with cardinality 0...1*, which associates between any class and a class typed of optional.

In (Keepence and Mannion 1999), the authors proposed the combination of patterns and discriminants to support representing of commonality and variability in software product line architecture. A *pattern* is represented by class and object diagrams. According to (Keepence and Mannion 1999), a *discriminant* is a feature that differentiates a system from another in a product family. They defined three types of discriminants: (i) *single discriminants*, which represent a set of mutually exclusive features; (ii) *multiple discriminants*, which represent a set of features which are not

mutually exclusive; and (iii) *optional discriminants*, which are features that may or may not be used.

The single discriminant represents an inheritance hierarchy that consists of a generic class called *base class* and a set of subclasses called *realm*. A realm is used to represent variants of a variation point in a product family. For the single discriminant, a subclass in a realm can be chosen for a product member. The multiple discriminant also represents an inheritance hierarchy that consists of a base class and realm. One or more subclasses in a realm can be chosen for a product member. The optional discriminant is represented by two classes with a 0..1 association.

3.5.3. Feature Modeling

This technique was initially proposed in FODA to assist the activity of domain analysis. As described in Section 3.4., many approaches apply and extend the definition of a feature model to support the development of product family systems. Thus, we describe below different aspects of the feature modeling technique that are applied in existing approaches.

I. Types of Features in a Feature Model

(i) *mandatory features* (Bosch 1998, Clements and Northrop 2002, Griss et al. 1998, Kang et al. 1990, Kang et al. 1998, PuLSE, Weiss 1995) are compulsory for product members in a family.

(ii) *optional features* (Bosch 1998, Clements and Northrop 2002, Griss et al. 1998, Kang et al. 1990, Kang et al. 1998, PuLSE, Svahnberg et al. 2001, Weiss 1995) may exist in a specific product member or not.

(iii) *alternative features* (Bosch 1998, Clements and Northrop 2002, Kang et al. 1990, Kang et al. 1998, PuLSE, Weiss 1995) or *variant features* (Griss et al. 1998), are a set of possible features that can be selected for a specific product member.

Moreover, (Svahnberg et al. 2001) define a feature type *external features* that is a feature unavailable in a system but needs to be satisfied by an external system.

II. Notations of Features in a Feature Model

As shown in Figure 3-2, a feature may be depicted as a round or a rectangle with its name inside. Many approaches applied the feature notation defined in (Kang et al. 1990). However, some approaches applied a UML class diagram for expressing features, for example (Griss et al. 1998). Moreover, different types of a feature i.e. mandatory, optional, and alternative are represented in different notations.

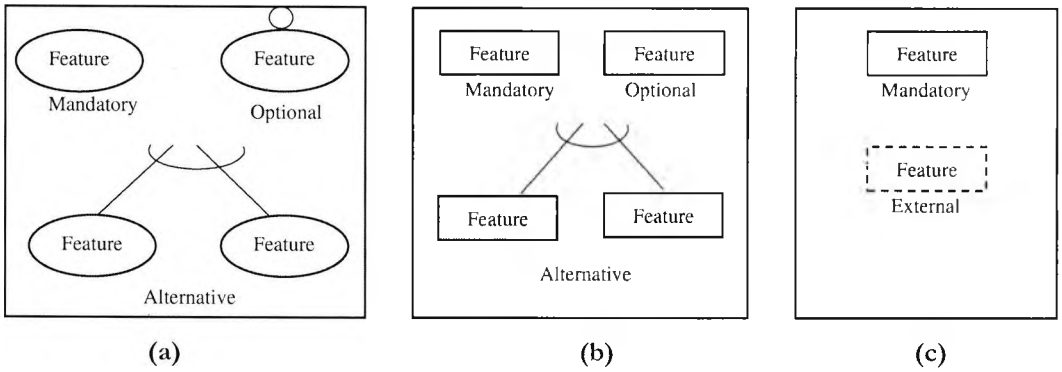


Figure 3- 2: different notations for different types of a feature: (a) (Kang et al. 1990); (b) (Griss et al. 1998, Kang et al. 1998); and (c) (Svahnberg et al. 2001)

III. Relationships between Features in a Feature Model

Ideally, features are atomic units that can be put together in a product without difficulty. However, features are generally not independent and several types of relations can exist between them. According to (Gibson et al. 1997), feature interaction is defined as a characteristic of a system whose complete behavior does not satisfy the separate specifications of all its features.

The types of relationships express the rules of feature interaction. These relationships are considered when features are selected for product members. They represent which features must be selected together and which features must not. Table 3-1 shows different types of relationships between features.

Table 3- 1 presents the classification of relationships between features:

Relationship type	Description
depends-on (Griss et al. 1998)	Indicating that a feature relies on an existence of another feature
mutually exclusive (Griss et al. 1998)	Indicating that two features must exist at the same time
conflicting (Griss et al. 1998)	Illustrating that related features have conflicting requirements.
composed-of (Kang et al. 1998), *composition* (Svahnberg et al. 2001)	Indicating that a feature is composed of other features
generalization/specialization (Kang et al. 1998), OR *specialization* (Svahnberg et al. 2001)	Indicating that a child feature is specialized from a parent feature
implemented-by (Kang et al. 1998)	Indicating that a feature is implemented by another feature
XOR *specialization* (Svahnberg et al. 2001)	Indicating that children features are mutually exclusive

3.5.4. Architecture Description and Component-based Languages

According to (America et al. 2000), the authors defined a component-based technique supports the design and implementation of software development. The technique includes the architecture description language, called *Koala* (Ommering et al. 2000). Koala is used to define a large diversity of product family systems. The Koala component model is created to represent architecture of product family. In Koala, a component consists of: (i) *interfaces*, which are communication ports between different components; (ii) *connectors*, which connect between different interfaces; and (iii) subcomponents, which are components in a particular component. They define two types of interfaces: (i) *provides interface*, which allows external components to use functionality implemented in a component; and (ii) *requires interface*, which allows a component to use functionality implemented in an

external component. They also define three types of connectors: (i) *direct connector*, which directly connect two interfaces; (ii) *switch,* which is a control for changing the direction of connection between components; and (iii) *module,* which is a process existing between two interfaces.

In the Koala component model, variation points are represented by using switch connectors. They describe that variation points are specialized during an activity of integration and testing in the application engineering. The Koala technique is applied in the COPA method that defines a framework of product family system development which is developed and used by Philips (Philips). All components in the Koala component model are implemented as source code components and connectors are provided for interconnection with other components.

Additionally, *xADL2.0* is a software architecture description language (ADL) that is developed by (xADL2.0). xADL is an extension of xArch (xArch) by applying with XML schemas. The language is compatible with XML technologies and XML tools. xADL is defined to provide the representation of variability in the domain of product family systems such as: (i) *optional elements*, which are parts of architecture such as components, connector, and interfaces; and (ii) *variants*, which provide a set of variable aspects of the architecture. Examples of approaches applied xADL 2.0 for product family system development are (Bastarrica et al. 2006, Westhuizen and Hoek 2002).

As described in Section 3.4, existing component-based techniques e.g. *commercial Off-The-Shelf* (COTS) (Dean 2002), Microsoft's COM (COM), Sun's JavaBeans (JavaBeans), COBRA (COBRA) are used in the activities during domain implementation. According to the literature (America et al. 2000, Atkinson et al. 2000, Bayer and Widen 2002, Coriat et al. 2000), the authors proposed to use existing component-based techniques for the development of reusable components in an activity of domain implementation. In addition to (Redondo et al. 2004), the authors defined a component-based approach for supporting an activity of domain analysis in software product line engineering. The approach applied the formal and

incremental method to represent requirements in the domain of product family systems. The requirements are grouped as components and specified a set of variants.

3.6. Supporting Tools for Product Family Systems

At this stage, tools for software product family system development rarely exist. Although some existing approaches for product family system development have provided tool support, most of the tools do not support the whole process of product family system development. Examples of these tools are such as Koala compiler (America et al. 2000), KoalaMaker (America et al. 2000), PuLSE-BEAT (Schmid and Schank 2000), DIVERSITY/CDA (Bayer et al. 1999), PASTA process modeling tool (Weiss 1995), and ASADAL (ASADAL). Additionally, some tools have been developed by projects and companies. Examples of these tools are such as V-Manage (ESAPS), MetaEdit+ (Metacase), and Product Line Platform (GEARS). Some commercial requirements engineering and general tools are applied and adopted in some existing approaches and projects. One example is that CAFA applied commercial tools such as DOORS (DOORS), RequistePro (RequisitePro), Rational Rose (RationalRose), Visio (Visio 2003), and XMI Toolkit (XMLToolKit) for software product line engineering. In (Gomaa and Shin 2004), they applied the commercial case tools i.e. Rational Rose and Rose RT (RationalRose) as well as integrate with their customised tool KBRET to support the design and generation of the proposed architecture for product family systems. In (Lago et al. 2004), they extended the commercial tool Together® ControlCenter™ (Borland) to develop and maintain design artefacts of a product family. However those tools in current literature still have limitations in terms of technique- and are platform-dependent.

The tools for product family systems are required to support the whole life cycle of product family system development. As mentioned before, existing tools are only focused on particular activities such as commonality and variability analysis and development of core assets. Other activities such as the production of product members from core assets, traceability between different artefacts, and evolution and maintenance are also required supporting tools.

Table 3-2 presents the comparison between existing approaches for product family system development by concerning with activities in domain engineering, activities in application engineering, artefacts created during these activities, techniques used, tool support provided, maturity of the approaches, and domain of the approaches.

Table 3- 2: Comparison approaches for product family system development

Approaches and Methods	Activities: Domain engineering	Activities: Application engineering	Generated artefacts	Techniques	Tool support	Maturity	Domains
COPA	Product family engineering; Platform engineering	Product engineering	Business models, design models, requirements, platform and product-specific components, Product line architecture	UML modeling, Use case specification, COTS, Microsoft's COM, Sun's JavaBeans, OMG's COBRA, Koala ADL	Koala compiler, KoalaMaker, Code editor + plugs-in, Visio	Have been applied by successful industries (Phillips)	Telecommunication, Medical imaging, Consumer electronics
QADA	Requirements engineering; Conceptual architecture design; Conceptual architecture analysis; Concrete architecture design; Concrete architecture analysis	N/A	Requirements, context model, decomposition model, allocation model, collaboration model, design rationale, knowledge base, structural diagrams, state diagrams, message sequence charts, deployment model, scenarios	UML modeling	Commercial UML, Visio, Word processing	Result from the research project PLANA (2001)	Information systems, Middleware, Wireless services

Approaches and Methods	Activities: Domain engineering	Activities: Application engineering	Generated artefacts	Techniques	Tool support	Maturity	Domains
KobrA	Context realization; Komponent specification; Komponent realization; Component reuse	Context realization instantiation; Framework instantiation	Komponents, object models	UML modeling, component-based techniques	Commercial UML, Word processing, Configuration management	Instance of PuLSE	Information systems
PuLSE	Initialisation; Infrastructure construction; Evolution and management	Infrastructure usage	PuLSE-BC, PuLSE-Eco, PuLSE-CDA, PuLSE-DSSA, PuLSE-I, supporting components, product line architecture	Component-based techniques	PuLSE-BEAT, DIVERSITY/CDA	Have been applied by industrials	Merchandise information systems, Stock market data evaluation, CAD systems, Human comfort simulations and layout systems

Approaches and Methods	Activities: Domain engineering	Activities: Application engineering	Generated artefacts	Techniques	Tool support	Maturity	Domains
SPLIT	Domain engineering	Application engineering	Requirements , software system product line architecture (SSPLA), patterns, guidelines, constraints, component models	UML modeling, COTS, component-based techniques	Commercial tools	Result from research projects (ESAPS and CAFÉ)	Telecommunication
FAST	Domain engineering	Applications engineering	Requirements	Domain specific language AML	PASTA process modeling tool		Real-time systems
RSEB	Requirements Engineering; Architectural family engineering; Component system engineering	Application system engineering	Requirements, product line architecture	Use case specification, UML modeling	N/A	N/A	N/A

Approaches and Methods	Activities: Domain engineering	Activities: Application engineering	Generated artefacts	Techniques	Tool support	Maturity	Domains
FODA	Domain analysis; Feature analysis; Feature modeling	**N/A**	Requirements, context model, feature model	Feature modeling	N/A	Have been applied and extended by researches and industrials since year 1990	N/A
FORM	Feature modeling; Architecture modeling; Component engineering	N/A	Requirements, feature model, subsystem model, process model, module model	Feature modeling	ASADAL	Extension of FODA and have been applied by industries	Electronic bulletin board, Private Branch Exchange, Elevator Control Systems
FeatuRSEB	Requirements Engineering; Architectural family engineering; Component system engineering	Application system engineering	Requirements, product line architecture	Use case specification, UML modeling, feature modeling	N/A	Extension of RSEB	N/A

3.7. Traceability of Product Family Systems

As suggested in (Boehm et al. 2004, Streitferdt 2001), traceability of product family systems is important due to some reasons:

(a) Traceability throughout artefacts is a necessary precondition for preserving the consistency of a product family during development and for software development in general;

(b) Traceability activity assists the reuse of artefacts across the product family. The benefits of product family approach depend on how effective an organisation can reuse the core assets of a family. When the percent of reuse is high, the cost of product member development is relatively low; and

(c) Traceability relations assist stakeholders to valid and verify software artefacts in product family.

Generally, traceability relations are used in different proposed: (a) impact analysis of a system; (b) validation and verifying software artefacts; (c) reuse of existing software artefacts; and (d) understanding software artefacts (as described in Section 2.5). According to the literature, establishing traceability relations in product family systems is expected to solve some of existing problems described in Section 3.2. Particularly, the use of traceability relations can (a) improve the communication between stakeholders; (b) assist defining commonality and variability due to different perspectives and lack of knowledge by using traceability information; (c) assist the documentation by applying with traceability information; and (d) decrease the confliction between artefacts by applying with traceability relations.

3.7.1. Existing Approaches for Traceability Generation in Product Family Systems

Software traceability can be established between the core assets created during product family development. According to Figure 3-1, traceability relations can be generated between reference requirements, software product line architecture, and reusable components. In addition, many traceability approaches (as described in

Chapter 2) are applied for the software system development. A few of them have been extended to satisfy the traceability activities in the domain of product family. In (Mohan and Ramesh 2002), the authors adopted the traceability reference model developed by (Ramesh and Jarke 2001) and applied the model with the e-services family systems.

Additionally, Bayer at el. (Bayer and Widen 2002) suggested the traceability activity into the PuLSE method. As described in Chapter 2, the authors defined the traceability reference model which consists of artefact types and relationship types in the domain of product family systems. More specifically, the model is focused on the artefacts created during domain engineering process. However, the authors do not explicitly define how to achieve the traceability activities or what tool supports is provided.

In (Berg and Bishop 2005), the authors defined the conceptual model for variability in product family systems. The model contains three types of artefacts: (i) *requirements*, represented in use cases and feature model; *architecture/ design,* represented in UML class diagrams; and (iii) *source code*, implemented in C++ programming language. The authors suggested traceability between variability in different types of artefacts. However, they do not define the classification of traceability relations and do not discuss of how to establish the relations. Additionally, the work in (Riebisch and Philippow 2001) has been proposed to use traceability relations for supporting the activities of product family system development. In other words, the traceability relations are used to identify possible reused artefacts of existing systems for new product members. The authors suggested the generation of traceability relations (a) between requirements and feature implementation, (b) between design and decision, and (c) between requirements, design decisions, and features. However, the reference model and classification of traceability relations are not explicitly defined. The authors also suggested the technique in (Dick 1999) for traceability activities; however, do not define any tool support for the activities. In this work, the traceability relations are

defined in the coarse-grained level and not cover all types of artefacts in the product family systems.

According to (Lago et al. 2004), the authors defined the model, called *simplified representation model*, which includes of artefacts and traceability relationships in the domain of product family systems; though the traceability relations are defined in the coarse-grained level. However, they do not explicitly define which techniques are used for traceability activities and do not provide tool support for the activities.

Additionally, the projects (CAFE 2003, ESAPS) have been concerned with the traceability activities. The traceability meta-model is defined; however, it does not represent particular types of traceability relations and do not provide any tool support for the activities. Although, the SPLIT/Cloud (Coriat et al. 2000) which is applied during the activity of domain analysis in the projects includes the generation of traceability relations. The method defined types of traceability relations created between functional and non-functional requirements and between requirements and constraints. There are no special tool support for traceability activities, even though the projects proposed to apply DOORS (DOORS) for supporting requirement engineering activities.

As described in Chapter 2, in (Kim et al. 2005), the authors defined a traceability reference model for the domain of product family systems. The model consists of artefacts types and traceability relationship types. However, the authors do not describe how to achieve the generation of traceability relations in an automatic way or explicitly define the tool support for the traceability activities.

In (Plankl and Bockle 2001), the authors suggested an approach for domain analysis in the software product line engineering and defined the requirements model including traceability relationships between requirements artefacts. However, they do not define how to achieve the activities.

3.7.2. Issues of Traceability Activities in Product Family Systems

There are two main issues that need to be tackled:

Difficulty to Establish Traceability across the Large Size and Diversity of Software Artefacts in Product Family Systems

It is difficult to establish traceability relations in the domain of product family systems due to the large size and diversity of software artefacts. Additionally, it is difficult to capture the semantics of traceability relations between those artefacts. This issue is also concerned in some approaches for traceability generation. The examples of these approaches are such as (Alexander 2003, Knethen et al. 2002). The authors have discussed that their approaches do not support the generation of traceability relations in the domain of product family systems.

Inadequate Tool Support

Although traceability is recognized as an important activity in the software development, the support of the traceability generation in the product family systems is still rare. Some approaches (Bayer and Widen 2002, Coriat et al. 2000, Lago et al. 2004) proposed a traceability reference model in the domain of product family systems; however, they do not provide automatic support for generating traceability relations. In addition, although some approaches proposed to tackle the traceability issue in the domain of product family systems, they do not provide automatic support (Bayer et al. 1999, CAFE 2003, Coriat et al. 2000, ESAPS, Hull et al. 2002, Kim et al. 2005, Plankl and Bockle 2001, Riebisch and Philippow 2001).

3.8. Summary

This chapter has provided background information for product family systems. It has presented the terminologies, existing problems, current approaches, current techniques and current tools in the domain of product family. It also illustrated the existing approaches and problems of traceability practice in the domain of product family systems. In the next chapter, we present a traceability reference model for product family systems.

Part II: The Approach

Chapter 4

Traceability Reference Model

This chapter describes a traceability reference model for product family systems. The reference model includes two main essentials. Firstly, the types of documents represented software artefacts of product family systems are described in Section 4.2. Secondly, the classification of relationships between those documents is defined in Section 4.3. The examples of traceability relations between software artefacts of product family systems and the summary of traceability reference model are also given in Section 4.3. Section 4.4 summarises of the chapter.

4.1. Introduction

As discussed in Chapter 1, we believe that a feature-based object-oriented engineering approach is required when developing product family systems. A feature-based approach is important to support domain analysis and domain design, enhance communication between customers and developers in terms of product features, and assist with the development of software product line architecture. On the other hand, an object-oriented approach is necessary to assist with the development of the various product members. We propose to use an extension of the FORM (Feature-Oriented Reuse Method) methodology (Kang et al. 1998) due to its maturity, practicality, and extensibility characteristics.

Our work concentrates on documents generated by FORM methodology such as feature, subsystem, process, and module models, for the core assets; and object-oriented documents such as use case specifications, class diagrams, statechart diagrams, and sequence diagrams, for the product members.

Furthermore, our approach combines feature-oriented and object-oriented documents and, therefore, requires a common representation of these document types. In our approach, the documents are represented in XML. We have chosen XML as a basis for our approach due to several reasons: (a) XML has become the de facto language to support data interchange among heterogeneous tools and applications, (b) the existence of large number of applications that use XML to represent information internally or as a standard export format (e.g. Unisys XML exporter for Rational Rose (RationalRose), Borland Together (Borland), ArgoUML (ArgoUML)), and (c) to allow the use of XQuery (XQuery) as a standard way of expressing traceability rules. Moreover, the OMG promotes the use of the XML Metadata Interchange (XMI) (XMI) to enable interchange of metadata between modeling tools that are based on OMG-UML and metadata repositories. XMI integrates OMG-UML modeling standards with Meta Object Facilities (MOF) and XML-W3C standard.

For each document type used in our approach we have created XML Schemas as shown in Appendix A. The textual sentences in the XML documents are annotated with part-of-speech assignment by using a general purpose grammatical tagger called CLAWS (CLAWS). This grammatical tagger assumes the British National Corpus (Leech et al. 1994) and tags the text in a four-stage process. In the first stage, the text to be tagged is divided into words and sentences. In the second stage, initial POS-tags are assigned to the words based on a lexicon and tagging rules. In the third stage, the initial tags are revised to take into consideration the context of the words based on other rules. In the fourth stage, tags are disambiguated using Markov Model (Poritz 1998). CLAWS has an error-rate of 1.5%. A more detailed explanation of CLAWS can be found in (CLAWS).

4.2. Product Family Software Artefacts

The software artefacts generated during the development of product family systems are classified in two levels, namely *product line level* and *product member level*. Product line level is concerned with the software artefacts created during activities in domain

engineering, while product member level is concerned with software artefacts created during activities in application engineering to be composed as product members in product family systems.

In the product line level, the software artefacts are divided into three types based on the various phases in domain engineering of the product family system development. According to the literature (see Chapter 3), the first type, known as *reference requirements* is concerned with artefacts created during the domain analysis phase. The second type, known as *software product line architecture* is concerned with artefacts created during the design phase. The third type, known as *reusable software components* is concerned with artefacts created during the domain implementation phase.

In the product member level, the software artefacts are also divided into three types based on the various phases in application engineering of the product family system development. The artefacts in the product member level are as concerned as the artefacts in single software systems (Nuseibeh and Easterbrook 2000). The first type is concerned with artefacts known as *requirements specification.* The second type is concerned with artefacts such as *design models.* The third type is concerned with artefacts such as *source code.* According to the literature (Bayer and Widen 2002, Bosch and Hogstrom 2000, Clements and Northrop 2002, Nuseibeh and Easterbrook 2000, Weiss and Lai 1999), the composition and integration of those artefacts develop into product members.

The work presented in this thesis is concerned with software artefacts created during the domain analysis and design phases of product family system development according to Figure 3-1. More specifically, we classify the software artefacts of our concern into two dimensions. The first dimension includes the product line and product member levels, while the second dimension includes the two phases of the software development process. Table 4-1 presents the set of software artefacts used in our approach for each dimension.

The artefacts in Table 4-1 are comparable to the artefacts proposed for product level and product member level in the literature. For example, reference requirements created during the domain analysis phase is comparable to *feature model* (Kang et al. 1998) in Table 4-1; software product line architecture is comparable to *subsystem*, *process*, and *module models* (Kang et al. 1998) in Table 4-1; requirements specification in product member level is comparable to *use case* (Cockburn 1997) in Table 4-1; and design models in product member level is comparable to *class*, *statechart*, and *sequence diagrams* (UML).

Table 4- 1: Documents used in our approach

	Analysis	Design
Product-Line Level	Feature model	Subsystem model Process model Module model
Product Member Level	Use Case	Class diagram Statechart diagram Sequence diagram

In the following, we described each type of artefacts and present examples of these artefacts.

Feature Model

A *feature model* is a software artefact that describes the abstraction of domain knowledge obtained from domain experts such as system users, analysts, and system developers, as well as other sources such as books, user manuals, design documents, and source programs. It describes the common and variable aspects (features) of a family in a domain (Griss et al. 1998, Kang et al. 1998, Svahnberg et al. 2001).

The feature model proposed in FORM (Kang et al. 1998) is based on the feature model from FODA (Kang et al. 1990) enhanced with a textual specification for each feature. Therefore, a feature model has two main components: *a graphical hierarchy* of features and *textual specification*. Figure 4-1 presents an example of a

graphical hierarchy of feature for a mobile phone product family, while Figure 4-2 presents an example of a textual specification for feature *Text Messages* in Figure 4-1.

As shown in Figure 4-1, a feature is represented by a name and can be (i) *mandatory*, when it must exist in the applications in the domain; (ii) *optional*, when it is not necessary to be present in the applications in the domain; or (iii) *alternative*, when it can be selected for an application from a set of features that are related to the same parent feature in the hierarchy.

The features can be classified into four groups namely (i) *application capabilities*, signifying features that represent functional aspects of the applications (e.g. calling, connectivity, personal preference, and tool features); (ii) *operating environments*, signifying features that represent attributes of the environment in which product members are used and operated (e.g. network, input and output methods, and operating system features); (iii) *domain technologies*, signifying features that represent specific implementation and technological aspects of the applications in the domain (e.g. WAP and XHTML[2] browser types; specific Java application support like mobile media and wireless messaging application programming interface; SMTP, POP3, and IMAP4[3] network protocol features); and (iv) *implementation techniques*, signifying features that represent more general implementation and technological aspects of the applications, but not necessary specific for the domain (e.g. PGP and DES encryption methods; AMR, MIDI, and MP3 sound formats; and 3GPP and MPEG[4] video format features).

Feature can also be related by different types of relationships. Examples of these relationships are (i) *composed_of*, (ii) *generalisation/specialization*, and (iii) *implemented_by* relationship types.

---

[2] WAP: Wireless Application Protocol; XHTML: Extensible HyperText Markup Language.

[3] SMTP: Simple Mail Transfer protocol; POP3: Post Office Protocol; IMAP4: Internet Message Access Protocol.

[4] AMR: Adaptive Multi-Rate; MIDI: Musical Instrument Digital Interface; MP3: MPEG Audio Layer III; 3GPP: 3rd Generation Partnership Project; and MPEG: Moving Picture Experts Group.

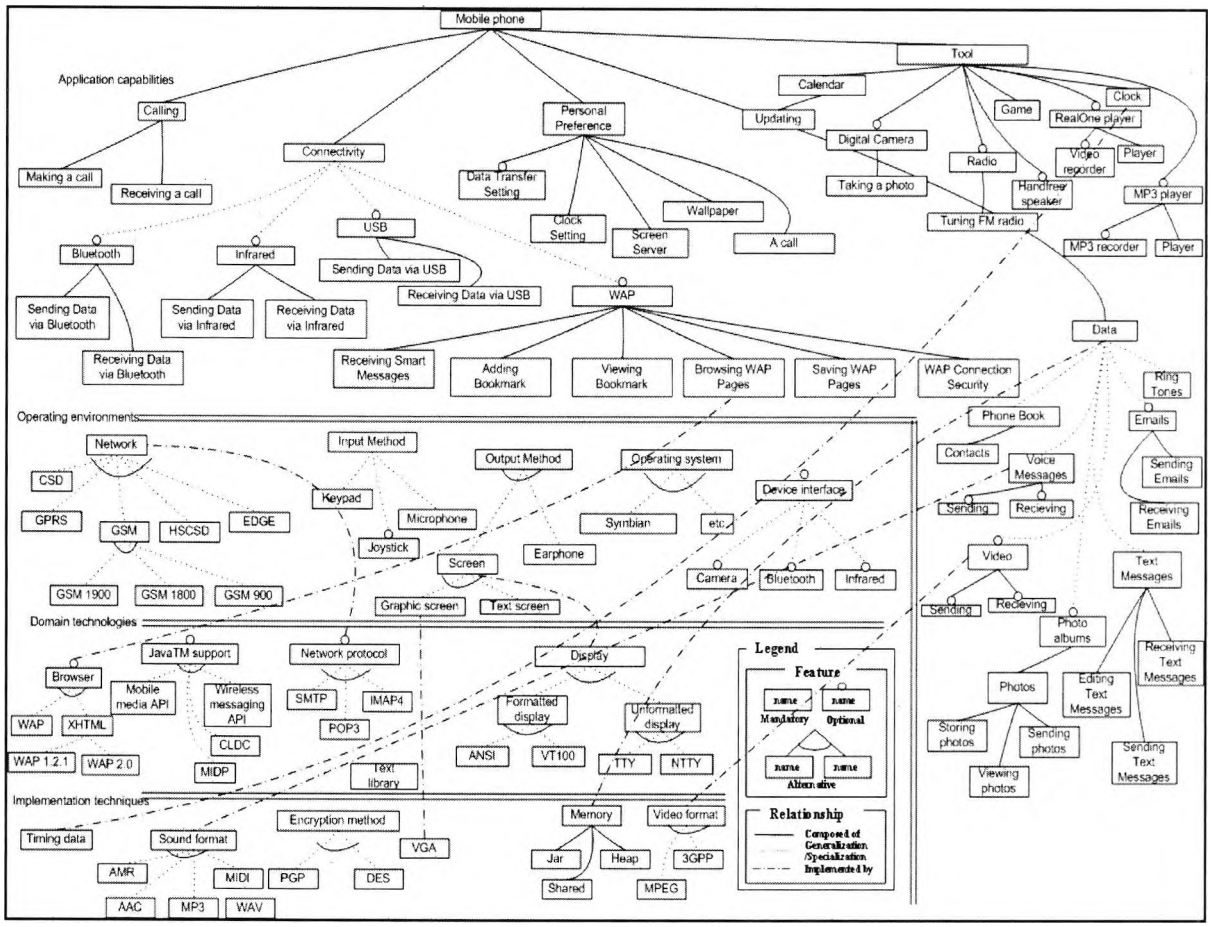


Figure 4- 1: The feature model of the mobile phone

As shown in Figure 4-2, the textual specification represents (i) a *name*, (ii) a *description*, (iii) *issues and decisions* representing trade-offs, rationale, or justifications for including the feature in an application, (iv) a *type* such as application capabilities, operating environments, domain technologies, and implementation technologies, (v) *commonality* indicating if a feature is mandatory, optional, and alternative, (vi) relationship with other features such as composed-of, implemented-by, generalisation/specialization, (vii) *composition rule* representing mutual dependency and mutual exclusion relationships to indicate consistency and completeness of a feature, if any, and (viii) *allocated-to-subsystem* indicating the name of a subsystem that contains the feature, if any

Feature-name:	Text Messages
Description:	The phone can edit, send, and receive a short text message
Issues and decision:	Text message over mobile phone is a way of communication
Type:	Application capability
Commonality:	Mandatory
Composed-of:	Sending Text Messages, Receiving Text Messages, Editing Text Messages
Composition-rule:	-
Allocated-to-subsystem:	Messaging

Figure 4- 2: Features in textual specification language (Kang et al. 1998).

We have developed an XML representation of the feature model based on the textual specification. In this representation, a feature model is composed of many features as shown in the extract of Figure 4. Each feature has a name (<Feature_name>), a description of the feature in natural language sentences (<Description>), a description of possible issues and decisions that may have been raised during the feature analysis process (<Issue_and_decision>), a type (<Type>), an element <Existential> denoting if the system is mandatory, optional, or alternative, relationships with other features (element <Relationship> with attribute *type* and the associated features represented in element <Rel_feature>), and the subsystem name that may contain the feature (<Allocated_to_subsystem>), if any. The contents of *Feature_name*, *Description*, *Issue_and_decision*, *Rel_feature*, and *Allocated_to_subsystem* elements are marked-up with part-of-speech XML tags (XML POS-tags) indicating their grammatical role in the sentence. For instance, the word "Text" is maked-up with element <NN1>, denoting that "Text" is a singular common noun; the word "Messages" is marked-up as <NN2>, denoting a plural common noun, the word "edit" is marked-up as <VVI> denoting an infinite verb.

```
<Feature_Model>
 <Feature>
 <Feature_name> <NN1> Text </NN1> <NN2> Messages </NN2> </Feature_name>
  <Description>
 <AT0> The </AT0> <NN1> phone </NN1> <VM0> can </VM0> <VVI> edit </VVI>
 <SC>,</SC> <VVI> send </VVI> <SC>,</SC> <CJC> and </CJC> <VVI> receive </VVI>
 <AT0> a </AT0> <AJ0> short </AJ0> <NN1> text </NN1> <NN1> message </NN1>
 <SC>.</SC>
  </Description>
  <Issue_and_decision>
    <NN1> Text </NN1> <NN1> message </NN1> <II> over </II> <JJ> mobile </JJ>
    <NN1> phone </NN1> <VBZ> is </VBZ> <AT1> a </AT1> <NN1> way </NN1>
    <IO> of </IO> <NN1> communication </NN1>
  </Issue_and_decision>
  <Type>Application capability</Type>
 <Existential>Mandatory</Existential>
 <Relationship Type="composed_of">
  <Rel_feature> <VVG> Sending </VVG> <NN1> Text </NN1> <NN2> Messages </NN2>
  </Rel_feature>
  <Rel_feature> <VVG> Receiving </VVG> <NN1> Text </NN1> <NN2> Messages </NN2>
  </Rel_feature>
 <Rel_feature> <VVG> Editing </VVG> <NN1> Text </NN1> <NN2> Messages </NN2>
 </Rel_feature>
 </Relationship>
 <Allocated_to_subsystem> <NN1> Messaging </NN1> </Allocated_to_subsystem>
</Feature>
<Feature>
   <Feature_name> <VVG> Editing </VVG><NN1> Text </NN1><NN2> Messages </NN2>
   </Feature_name>
     <Description>
      <AT0> The </AT0> <NN1> phone </NN1> <VVZ> provides </VVZ> <AT1>an</AT1>
      <NN1> editor </NN1> <TO> to </TO> <VVI> create </VVI> <AT1> a </AT1>
      <JJ> new </JJ> <NN1> text </NN1> <NN1> message </NN1> <SC>.</SC>
      <VVG> Editing </VVG> <AT> the </AT> <NN1> text</NN1> <NN1> message </NN1>
      <VM> can </VM> <VBI> be </VBI> <VDN> done </VDN> <II> in </II>
       <JJ> different </JJ> <NN2> ways </NN2> <II> such </II> <II> as </II>
       <NN1> alpha </NN1> <NN1> mode </NN1> <CC> and </CC> <JJ> predictive </JJ>
       <NN1> mode </NN1>
    </Description>
   <Type>Application capability</Type>
   <Existential>Mandatory</Existential>
   <Allocated_to_subsystem> <NN1> Messaging </NN1> </Allocated_to_subsystem>
 <Composition_rule>
    <VVZ> requires </VVZ> <AT> the </AT> <NN1> text </NN1> <NN1> library </NN1>
    <NN1> feature </NN1>
    </Composition_rule>
 </Feature>
  ...
</Feature_Model>
```

Figure 4- 3: Feature model for mobile phone systems

Use Cases

Use case is a textual specification language that captures a contract between the stakeholders of a system about its behavior (Cockburn 1997). In our work, we represent the functional requirements of product members as use-cases based on the template proposed in (Cockburn 1997).

In our template, a use case is represented by element (<Use_Case>) with a unique identifier (*UseCaseID*), information about the product line domain (*System*) and product member identifier (*Product_Member).* A use case has also a title (<Title>)*;* a brief textual description (<Description>)*;* the level of functionality that it describes within a system (<Level>); pre- and post-conditions that must be satisfied before and after its execution respectively (<Preconditions> and <Postconditions>)*;* primary and secondary actors describing the users of the use case (<Primary_actor> and <Secondary_actors>); flow of events denoting the events that trigger the use case and the specification of the normal events that occur within it (<Flow_of_events>); exceptional events describing the events that not always occur when the use case is executed (<Exceptional_events>)*;* and Superordinate and subordinate use cases (<Superordinate_use_case> and <Subordinate_use_case>). As in the case of feature model, the words in the textual parts of the use case are annotated with XML POS-tags denoting their grammatical roles.

Figure 4-4 illustrates an example of a use case *Sending a Message* from a mobile phone for product member PM1 of the mobile phone case study. The use case (<Use_Case>) is identified with *UseCaseID* ("UC1"), *System* ("MobilePhone"), and *Product_Member* ("PM1") It contains elements i.e. <Title>, <Description>, <Level>, <Preconditions>, <Postconditions>, <Primary_actor>, <Secondary_actors>, <Flow_of_events>, <Event>, <Exceptional_events>, <Superordinate_use_case>, and <Subordinate_use_case> that describe the context of the use case. Likewise, the elements that are composed of textual descriptions are marked-up with part-of-speech indicating their grammatical role in the sentence as XML tags.

```
<Use_Case UseCaseID="UC1" System="MobilePhone" Product_Member="PM1" >
 <Title> Sending a Message </Title>
 <Description> <AT0> The </AT0> <NN1> phone </NN1> <VBZ> is </VBZ>
      <AJ0> able </AJ0> <TO0> to </TO0> <VVI> send </VVI> <AT0> a </AT0>
      <NN1> text </NN1> <NN1> message </NN1> <SC> . </SC> <AT> The </AT>
      <NN1> user </NN1> <VM> can </VM> <VVI> specify </VVI> <AT1> an </AT1>
      <NN1> address </NN1> <IO> of </IO> <AT1> a </AT1> <NN1> receiver </NN1>
      <II> by </II> <VVG> selecting </VVG> <II> from </II> <AT1> a </AT1>
      <NN1> list </NN1> <IO> of </IO> <NN2> contacts </NN2> <SC> . </SC>
</Description>
  <Level> User Goal </Level>
  <Preconditions> <AT> The </AT> <NN1> user </NN1> <VHZ> has </VHZ>
      <VHZ> already </VHZ> <VVN> selected </VVN> <NN1> function </NN1>
      <IO> of </IO> <VVG> sending </VVG> <AT1> a </AT1> <NN1> text </NN1>
      <NN1> message </NN1> <II> from </II> <AT> the </AT> <JJ> main </JJ>
      <NN1> menu </NN1> <SC> . </SC>
</Preconditions>
  <Postconditions> <AT> The </AT> <NN1> phone </NN1> <VHZ> has </VHZ>
            <VVN> sent </VVN> <AT> the </AT> <NN1> message </NN1> <SC> . </SC>
  </Postconditions>
  <Primary_actor> The user </Primary_actor>
  <Secondary_actors/>
  <Flow_of_events>
    <Event> <AT> The </AT> <NN1> system </NN1> <VVZ> shows </VVZ>
      <AT1> an </AT1> <NN1> editor </NN1> <IF> for </IF> <VVG> writing </VVG>
      <AT1> a </AT1> <NN1> message </NN1> <SC> . </SC> </Event>
    <Event> <AT> The </AT> <NN1> user </NN1> <VVI> type </VVI> <AT1> a </AT1>
      <NN1> phone </NN1> <NN1> number </NN1> <IO> of </IO> <AT1> a </AT1>
      <NN1> receiver </NN1> <SC> . </SC> <AT>The</AT> <NN1> user </NN1>
      <VVI> select </VVI> <AT1> a </AT1> <NN1> phone </NN1> <NN1> number </NN1>
      <IO> of </IO> <AT1> a </AT1> <NN1> receiver </NN1> <II> by </II>
      <VVG> selecting </VVG> <II> from </II> <AT1> a </AT1> <NN1> list </NN1>
      <IO> of </IO> <NN2> contacts </NN2> <SC> . </SC> <AT> The </AT>
      <NN1> user </NN1> <VVI> send </VVI> <AT> the </AT> <NN1> text </NN1>
      <NN1> message </NN1> <II> to </II> <JJ> multiple </JJ> <NN2> receivers </NN2>
      <II> by </II> <VVG> inserting </VVG> <JJ> multiple </JJ> <JJ> mobile </JJ>
      <NN1> phone </NN1> <NN2> numbers </NN2> <SC>.</SC> </Event>
    <Event> <AT> The </AT> <NN1> system </NN1> <VVD> displayed </VVD>
        <AT> the </AT> <NN1> phone </NN1> <NN1> number </NN1> <SC> . </SC>
    </Event>
    <Event> <AT> The </AT> <NN1> user </NN1> <VVI> enter </VVI> <AT> the </AT>
        <NN1> message </NN1> <SC> . </SC> ... </Event>
    ....
  </Flow_of_events>
  <Exceptional_events/>
  <Superordinate_use_case/>
  <Subordinate_use_case/>
</Use_Case>
```

Figure 4- 4: Use case *Sending a Message*

Subsystem Model

In FORM, the *subsystem model* is a graphical diagram that represents high-level abstraction of the software product line architecture. Figure 4-5 shows an example of subsystem model for the mobile-phone product family case study. It illustrates functional groups of software, named as *subsystem*(s) and their interactions. A subsystem can be *internal*, when it exists in the product family system or *external*, when it does not belong to the product family system, but interacts with the internal subsystems of the family.

An interaction represents how the subsystems communicate with each other. There are two types of interactions i) *data flow*, representing a flow of data between subsystems and ii) *control flow*, representing a flow of control between subsystems. As shown in Figure 4-5, a subsystem model of the mobile phone product family consists of five subsystems, namely *operating*, *messaging*, *mobile Internet*, *network*, and *calling and house-in applications* subsystems.

We propose an XML representation for a subsystem model, as shown in Figure 4-6. In our template, a subsystem model for a domain is composed of various subsystems. Each subsystem has a name (<Subsystem_name>), has a brief textual description (<Description>), and can be of type internal or external (<Type>). The data and control flows between the subsystems are represented by element <Flow> with attributes denoting the unique identifier (*flow_id*), the type (*flow_type*), and the subsystems sending and receiving the flow (*sender* and *receiver*, respectively). The complete XML representation of the subsystem model with annotated XML POS-tags can be found in Appendix A.

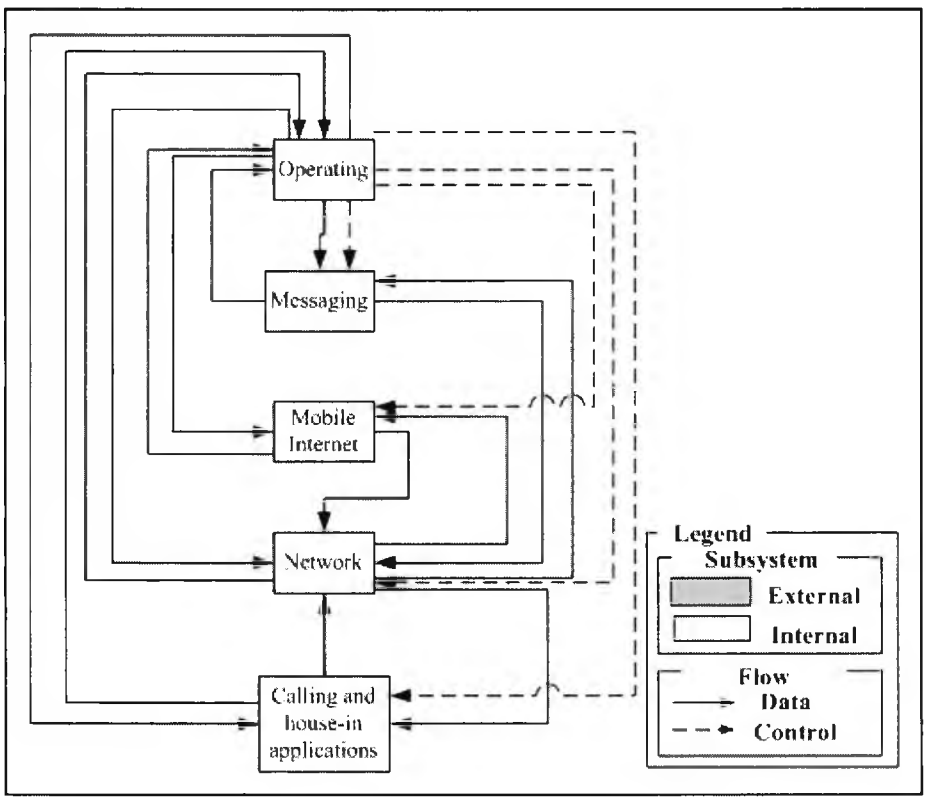


Figure 4- 5: Subsystem model of mobile phone systems

Figure 4-6 shows an extract of the subsystem model from a mobile phone product family. The extract of the subsystem model (<Subsystem_Model>) includes two subsystems (<Subsystem>), namely *Operating* and *Messaging*. The description (<Description>) of a subsystem specified in natural language sentences are represented with annotated XML POS-tags. Both two subsystems are typed of *internal*. Moreover, the extract of the subsystem model includes two flows (<Flow>), identified with *c1* and *d2*. Flow *c1* is typed of *control_flow* and sent from *Operating* subsystem to *Messaging* subsystem, while flow *d2* is typed of *data_flow* and sent from *Messaging* subsystem to *Mobile Internet* subsystem.

```
<Subsystem_Model>
    <Subsystem>
    <Subsystem_name> Operating </Subsystem_name>
      <Description> <DD1> The </DD1> <NN1> subsystem </NN1> <VVZ> provides </VVZ>
        <NN2> facilities </NN2> <IF> for </IF> <VVG> performing </VVG>
        <JJ> basic </JJ> <NN2> tasks </NN2> <II> such </II> <II> as </II>
        <NN1> control </NN1> <IO> if </IO> <AT> the </AT> <NN1> interaction </NN1>
        <IW> with </IW> <DB> all </DB> <NN1> control </NN1> <IO> of </IO>
        <AT> the </AT> <NN1> interaction </NN1> <IW> with </IW> <DB> all </DB>
        <NN2> devices </NN2> <SC> , </SC> <NN1> software </NN1> <SC> , </SC>
        <CC> and </CC> <NN0> data </NN0> <SC> ; </SC> <NN1> support </NN1>
        <IO> of </IO> <AT> the </AT> <NN1> interaction </NN1> <II> between </II>
        <JJ> internal </JJ> <NN2> applications </NN2> <SC> ( </SC> <REX> e.g. </REX>
        <NN2> games </NN2> <SC> , </SC> <NN0> multimedia </NN0> <SC> , </SC>
        <CC> and </CC> <NN1> PC </NN1> <NN1> connective </NN1> <SC> ) </SC>
        <SC> , </SC> <NN1> recognition </NN1> <IO> of </IO> <JJ> internal </JJ>
        <NN1> hardware </NN1> <SC> ( </SC> <REX> e.g. </REX> <NN1> screen </NN1>
        <SC> , </SC> <NN1> keypad </NN1> <SC> , </SC> <CC> and </CC>
        <NP1> Bluetooth </NP1> <SC> ) </SC> <CC> and </CC> ...
    </Description>
    <Type> internal </Type>
    </Subsystem>
    <Subsystem>
    <Subsystem_name> Messaging </Subsystem_name>
    <Description> <DD1> The </DD1> <NN1> subsystem </NN1> <VVZ> manages </VVZ>
        <AT> the </AT> <NN1> exchange </NN1> <CC> and </CC>
        <NN1> manipulation </NN1> <IO> of </IO> <NN2> messages </NN2> <SC> . </SC>
        <PPH1> It </PPH1> <VVZ> supports </VVZ> <MC> two </MC>
        <NN2> services </NN2> <SC> : </SC> <JJ> short </JJ> <NN1> message </NN1>
        <NN1> service </NN1> <SC> ( </SC> <NP1> SMS </NP1> <SC> ) </SC>
        <IF> for </IF> <JJ> textual </JJ>  <NN2> messages </NN2> <SC> , </SC>
        <CC> and </CC> <NN> multimedia </NN> <NN1> message </NN1>
        <NN1> service </NN1> <SC> ( </SC> <NNU2> MMS </NNU2> <SC> ) </SC>
        <IF> for </IF> <NN> multimedia </NN> <NN2> messages </NN2> <SC> . </SC>
        <AT> The </AT> <NN2> services </NN2> <VBR> are </VBR> <VVN> based </VVN>
        <II> on </II> <AT1> a </AT1> <NN1> store </NN1> <CC> and </CC>
        <JJ> forward </JJ> <NN1> protocol </NN1> <SC> . </SC> <AT> The </AT>
        <NN1> subsystem </NN1> <VVZ> interacts </VVZ> <IW> with </IW>
        <JJ> short </JJ> <NN1> message </NN1> <NN1> service </NN1>
        <NN2> centers </NN2> <SC>  ( </SC> <NP1> SMSC </NP1> <SC> ) </SC>
        <CC> or </CC> ...
    </Description>
    <Type> internal </Type>
  </Subsystem>
  ...
  <Flow flow_id = "c1" flow_type = "control_flow" sender = "Operating"
        receiver = "Messaging"/>
  <Flow flow_id = "d2" flow_type = "data_flow" sender = "Messaging"
        receiver = "Mobile Internet"/>
  ...
</Subsystem_Model>
```

Figure 4- 6: Example of Subsystem Model

Process Model

FORM proposes each subsystem in the subsystem model to be refined by one or many *process model*(s). The process model is a graphical diagram that represents the middle-level of the software product line architecture as shown in Figure 4-7. The process model is composed of many processes that refine the behavior of a particular subsystem, many messages that represent the communication between processes, and shared data that may be used by the processes.

Each process can be categorized as *resident* or *transient* depending if the process belongs to the subsystem (Kang et al. 1998). The *resident* process is allocated in the subsystem and the *transient* process exists outside the subsystem, but appears having messages exchanging to a resident process in the model. Additionally, a process can also be classified as *multiple* or *single,* depending on the necessary number of instances of a process to perform a task. The process model also represents the messages exchanged between the various processes and the data shared by a process. The messages can be (i) *closely-coupled*, which supports synchronized communication (implemented by a protocol of *message/reply*), and (ii) *loosely-coupled*, which supports an asynchronized communication (implemented by *message queue*) (Gomaa 1993). Shared data includes database, reports, and files.

As shown in Figure 4-7, *short messaging service (SMS)* process model represents the processes that exist in a *Messaging* subsystem (as shown in Figure 4-5). The *SMS* process model includes resident processes (i.e. *edit* process, *control* process, *check signal* process, *notification* process, and *short messaging service (SMS) control* process) and multiple processes (i.e. *update remotely* process and *short messaging service center (SMSC)* process).

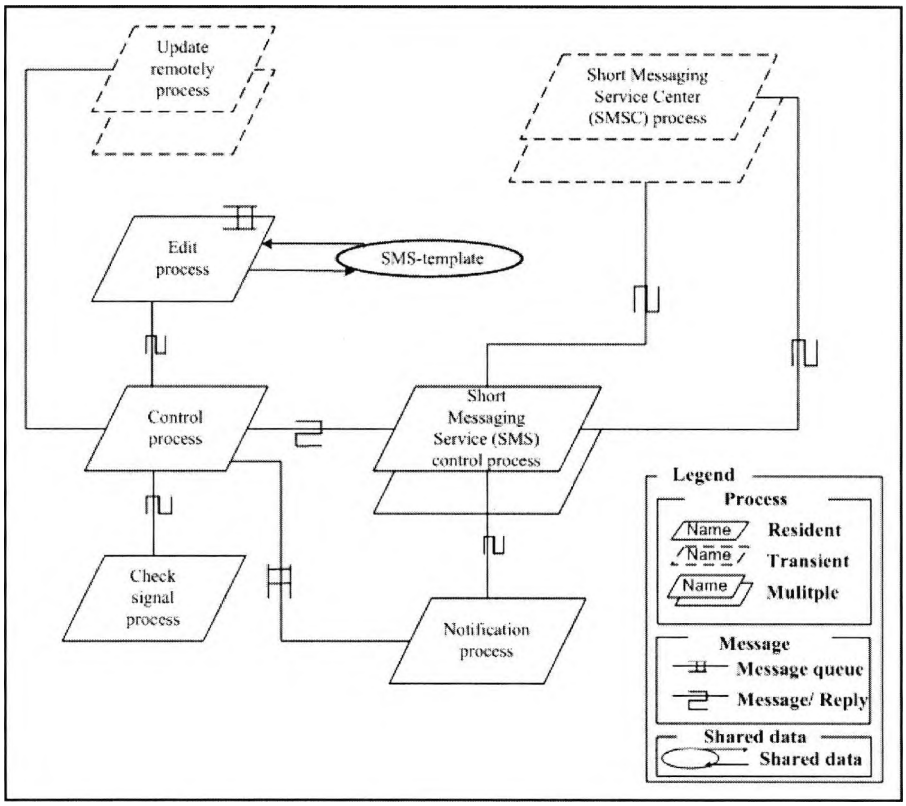


Figure 4- 7: *SMS* process model for *messaging* subsystem

We propose an XML representation for process models, as shown in Figure 4-8. The XML specification for process model used in our approach for *Messaging subsystem* is shown in Figure 4-7. A process model (<Process_Model>) has a unique identifier (*ProcessModelID*) and the name of the respective subsystem (*Subsystem_name*). Each process (<Process>) in the model may have an optional attribute referencing the shared data storage (*shared_data*). A process is specified by a name (<Process_name>), has a textual description (<Description>), can be concerned with multiple or single activities (<Activity>), and can be of type resident or transient (<Type>). The messages exchanged between the various processes (<Message>) are represented by a unique identifier (*message_id*); a type (*mesage_type*) that can be closely_coupled, when it supports synchronize communication, or loosely_coupled, when it supports asynchronous communication; and the processes receiving and sending the message (*sender* and *receiver*). The shared data storage

(<Shared_data>) has identifier and type attributes (*data_id* and *type*). Examples of process models in XML with XML POS tags are found in Appendix A.

```
<Process_Model ProcessModelID = "P1" Subsystem_name = "Messaging">
 <Process>
  <Process_name> Short Messaging Service (SMS) Control </Process_name>
  <Description> <AT> The </AT> <NN1> process </NN1> <WZ> performs </WZ>
      <AT> the </AT> <NN> delivery </NN> <CC> and </CC> <NN1> receiving </NN1>
      <IO> of </IO> <AT1> a </AT1> <JJ> short </JJ> <NN1> message </NN1> <II> to </II>
      <AT1> a </AT1> <JJ> short </JJ> <NN1> message </NN1> <NN1> service </NN1>
      <NN1> center </NN1> <SC> ( </SC)> <NP1> SMSC </NP1> <SC> ) </SC>
      <SC> . </SC> <AT> The </AT> <NP1> SMSC </NP1> <VBZ> is </VBZ>
      <VVN> connected </VVN> <II> to </II> <AT> the </AT>
      <NN1> telecommunication </NN1> <NN1> network</NN1>
      <SC> ( </SC> <REX> e.g. </REX> <NNU> GSM </NNU> <SC> , </SC>
      <NP1> HSCSD </NP1> <SC> , </SC> <CC> and </CC> <NN1> EDGE </NN1>
      <SC> ) </SC> <II> through </II> <AT> the </AT> <JJ> short </JJ>
      <NN1> message </NN1> <NN1> service </NN1> <NN1> gateway </NN1>
      <JJ> mobile </JJ> <JJ> switching </JJ> <NN1> center </NN1> <SC> ( </SC>
      <NP1> SMS </NP1> <NP1> GMSC </NP1> <SC> ) </SC> <SC> . </SC>
      <AT> The </AT> <NN1> process </NN1> <RR> also </RR> <VVZ> attaches </VVZ>
      <JJ> extra </JJ> <NN1> information </NN1> <II> about </II> <NP1> SMSC </NP1>
      <II> in </II> <AT1> a </AT1> <JJ> short </JJ> <NN1> message </NN1> <SC> . </SC>
     </Description>
<Activity>multiple</Activity>
<Type>resident</Type>
 </Process> ...
 <Process shared_data = "d1">
  <Process_name> Edit </Process_name>
  <Description> <DD1> This </DD1> <NN1> process </NN1> <VVZ> performs </VVZ>
   <AT> the </AT> <NN1> composition </NN1> <IO> of </IO> <AT1> a </AT1>
   <JJ> short </JJ> <NN1> message </NN1> <SC> . </SC> <AT> The </AT>
   <JJ> short </JJ> <NN1> message </NN1> <VVZ> contains </VVZ> <AT1> a </AT1>
   <NN2> receivers </NN2> <NN1> address </NN1> <CC> and </CC>
   <NN1> context </NN1> <SC> . </SC> <AT> The </AT> <NN1> process </NN1>
   <VVZ> provides </VVZ> <AT1> a </AT1> <NN1> list </NN1> <IO> of </IO>
   <NN2> contacts </NN2> <CC> and </CC> <AT1> a </AT1> <NN1> set </NN1>
   <IO> of </IO> <NN1> template </NN1> <JJ> short </JJ> <NN2> messages </NN2>
   <SC> . </SC> <AT> The </AT> <NN1> process </NN1> <VVZ> supports </VVZ>
   <MC> two </MC> <NN1> editing </NN1> <NN2> modes </NN2> <REX> i.e. </REX>
   <NN1> alpha </NN1> <NN1> mode <NN1> <CC> and </CC> <JJ> predictive </JJ>
   <NN1> mode <NN1> <SC> . </SC> <AT> The </AT> <NN1> alpha </NN1>
   <NN1> mode <NN1> <VVZ> accepts </VVZ> <JJ> alphanumeric </JJ> <SC> . </SC>
   <AT> The </AT> <JJ> predictive </JJ> <NN1> mode </NN1> <VVZ> predicts </VVZ>
   <AT1> a </AT1> <NN1> word </NN1> <II> from </II> <AT1> an </AT1>
   <NN1> input </NN1> <NN1> keystroke </NN1> <SC> . </SC> </Description>
<Activity>single</Activity>
<Type>resident</Type>
</Process> ...
<Message message_id="m7_trigger"  message_type="closely-coupled"
      sender="Short Messaging Service (SMS) Control"  receiver="Notification"/>
<Message message_id="m8_response"  message_type="closely-coupled"
      sender="Notification"  receiver="Short Messaging Service (SMS) Control"/> ...
<Shared_data data_id="d1" type="database"/>  ...
</Process_Model>
```

Figure 4- 8: SMS process model for *messaging* subsystem

Module Model

In FORM, each process in the process model is further refined in a *module model.* A module model is represented as a graphical diagram that represents the low-level of the software product line architecture. Figure 4-9 represents the module model for *Short Messaging Service SMS Control* process of *Messaging* subsystem.

A module model represents a hierarchical structure of the various modules composing a process and their interactions. The modules are classified into four groups related to the different groups of features (see earlier described) namely (i) *service modules*, which support the functionality of the systems and correspond to application capability features; (ii) *environment hiding modules*, which represent the running environment of the system and correspond to the operating environment features; (iii) *technique hiding modules*, which represent the technology domain aspects of the system and correspond to the domain technologies features; and (iv) *utility modules*, which represent general purpose aspects of the system and correspond to implementation techniques features.

We propose an XML representation for module models as shown in Figure 4-10. The XML specification for module model used in our approach for process *Short Messaging Service (SMS) Control* is shown in Figure 4-9. A module model (<Module_Model>) has a unique identifier (*ModuleModelID*), the corresponding process *(Process_name),* and is composed of various modules (<Modules>) and links (<Link>). Each module has a name (<Module_name>), a description (<Description>), and a type (<Type>). The type of a module is concerned with the detail of its code implementation ranging from *skeleton* (code outline), *template* (more detailed code without parameter specifications), to *precoded* (complete code). The links have a *type* (uses or inherits) and the *source* and *destination* modules. Examples of module models in XML with POS tags used in our approach are found in Appendix A.

Two modules can be associated by links. The methodology suggests two types of links: (i) *uses*, signifying that a module uses another module; and (ii) *inherits*, signifying that a module inherits another module.

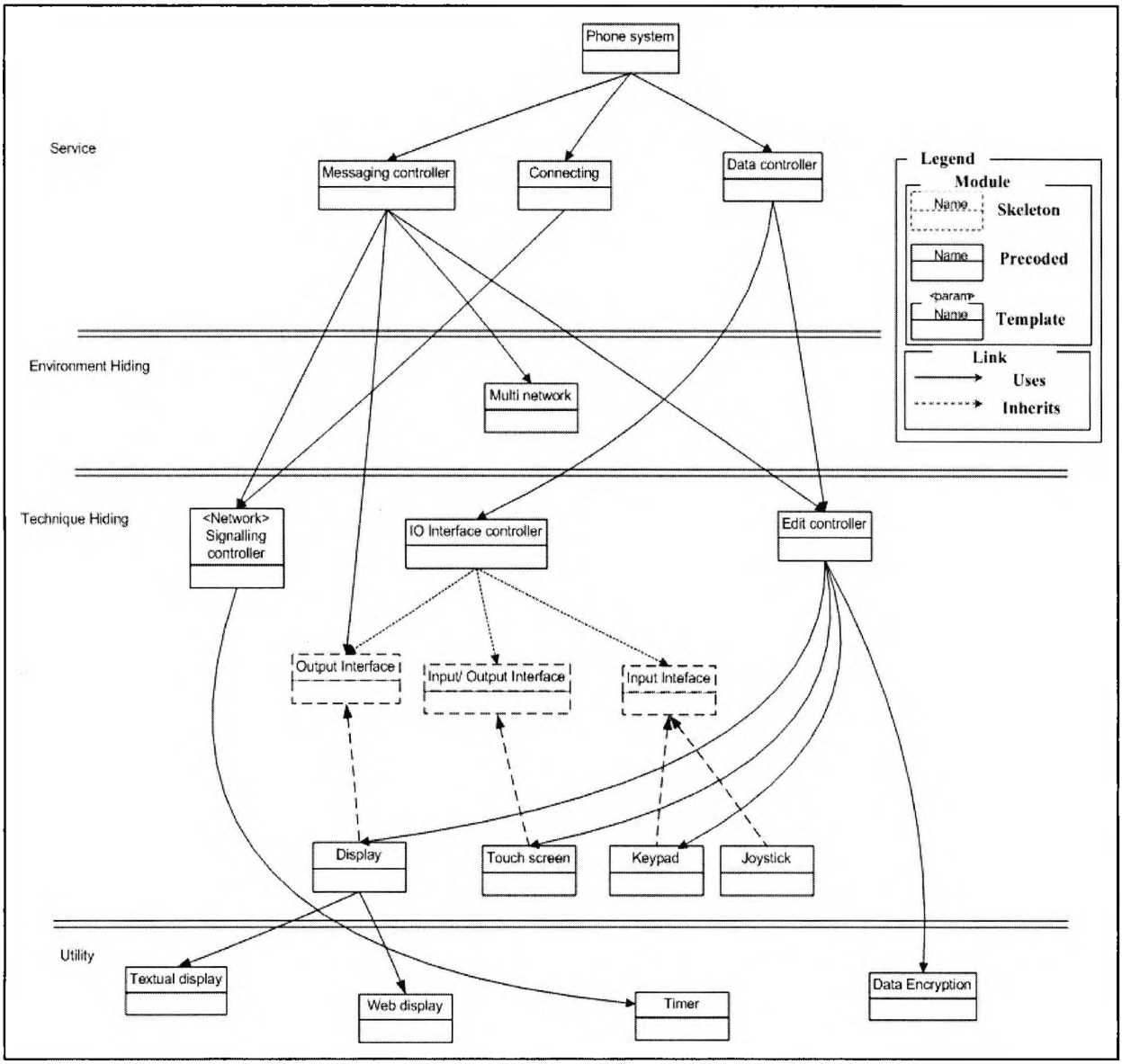


Figure 4- 9: The module model for *short messaging service (SMS) control* process of *messaging* subsystem

```
<Module_Model ModuleModelID = "MM1"  Process_name = "Short Messaging Service SMS
Control">
  <Module>
 <Module_name> Short Messaging </Module_name>
 <Description> <AT> The </AT> <JJ> maximum </JJ> <NN1> length </NN1> <IO> of </IO>
                <AT1> a </AT1> <NN1> text </NN1> <NN1> message </NN1>
                <VBZ> is </VBZ> <MC> 160 </MC> <NN2> characters </NN2>
                <SC> , </SC> <NN2> numbers </NN2> <SC> , </SC> <CC> or </CC>
                <DD> any </DD> <JJ> alphanumeric </JJ> <NN1> combination </NN1>
                <SC> . </SC> <DD1> This </DD1> <NN1> module </NN1>
                <RR> also </RR> <VVZ> supports </VVZ> <IF> for </IF>
                <NN1> non-text </NN1> <VVN> based </VVN> <JJ> short </JJ>
                <NN2> messages </NN2> <II21> such </II21> <II22> as </II22>
                <JJ> binary </JJ> <NN1> format </NN1> <DDQ> which </DDQ>
                <SC> , </SC> <VBZ> is </VBZ> <VVN> used </VVN> <IF> for </IF>
                <NN1> ring </NN1> <NN2> tone </NN2> <CC> and </CC>
                <NN2> logo </NN2> <NN2> services </NN2> <SC> . </SC>
                ...
 </Description>
 <Type> precoded </Type>
  </Module>
  ...
  <Link type="inherit" source="Short Messaging" destination="Messaging Edit"/>
  ...
 </Module_Model>
```

Figure 4- 10: Module model for *short messaging service SMS control* process

Class, Statechart, and Sequence Diagrams

The design of product members are described in UML class, statechart, and sequence diagrams. In our approach, these diagrams are represented in XMI format (XMI). We present here extracts of a class diagram (Figure 4-11), statechart diagram (Figure 4-12), and sequence diagram (Figure 4-13) used in our case study.

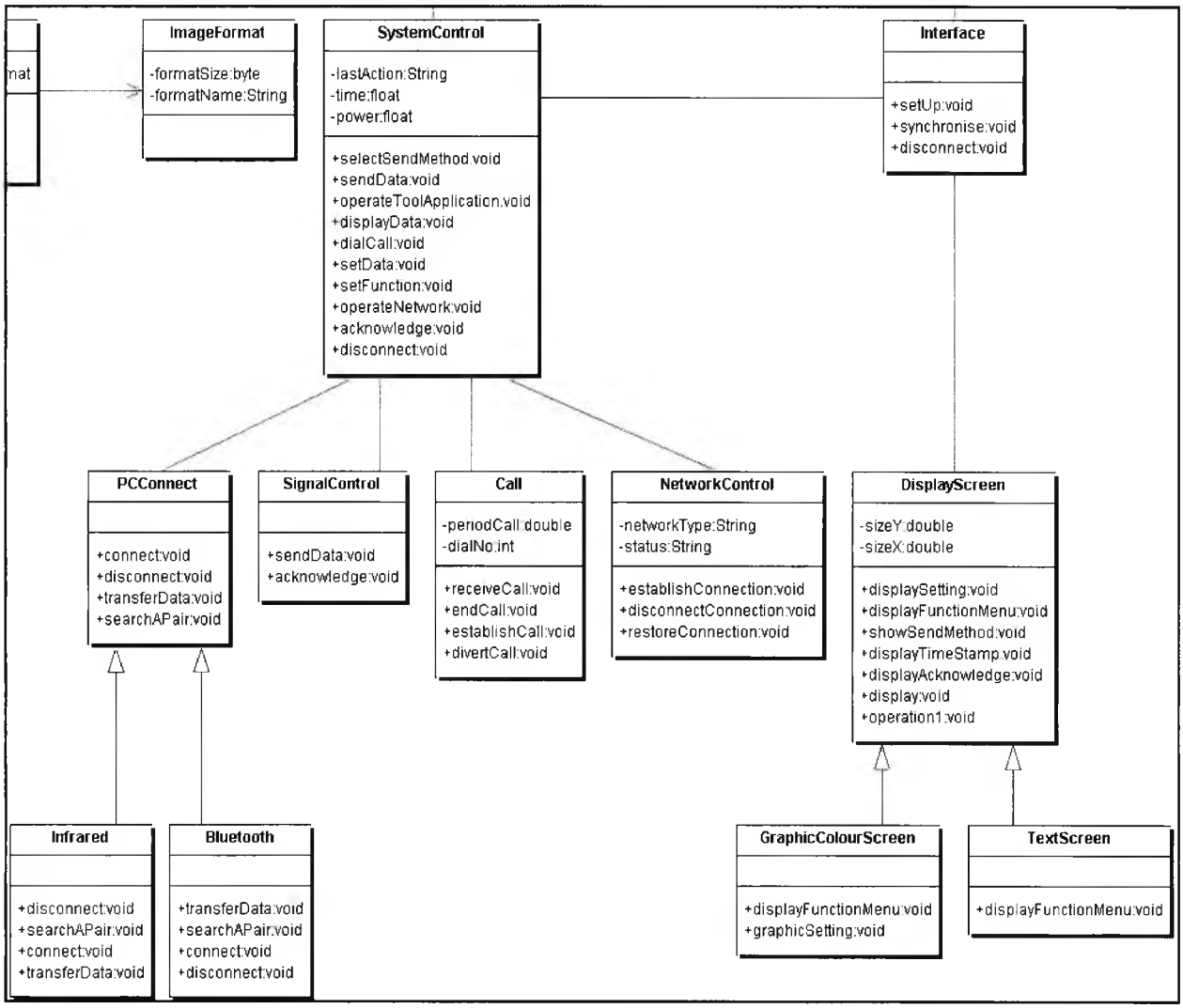


Figure 4- 11: An extract of a class diagram for product member *PM1*

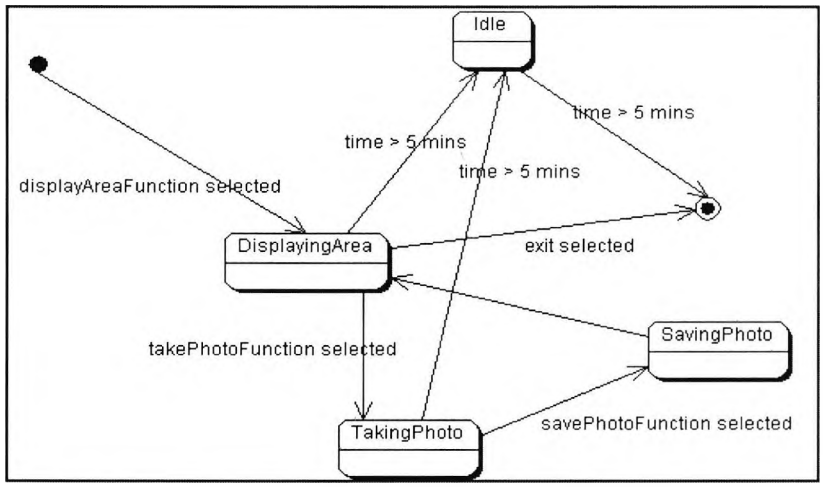


Figure 4- 12: A statechart diagram for a *digital_camera* class

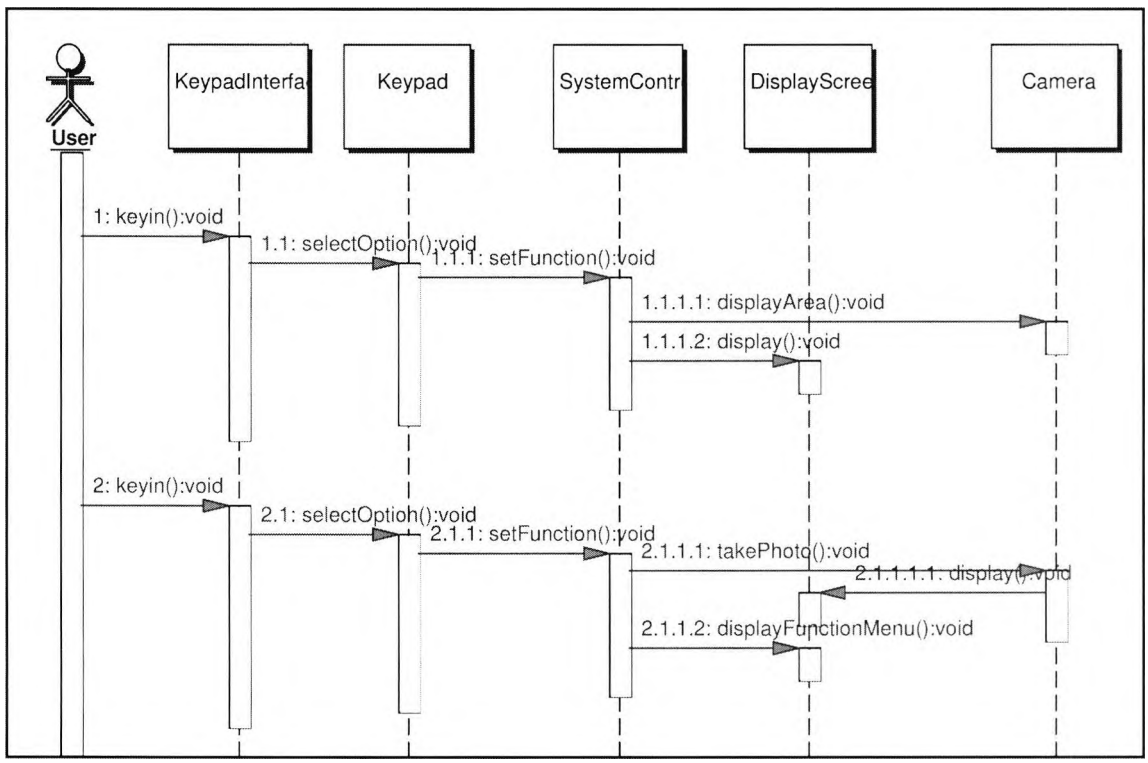


Figure 4- 13: An extract of a sequence diagram of *taking a photo*

4.3. Traceability Relations

Based on our study and analysis of the mobile phone domain, our study and experience with software traceability, the types of traceability relations proposed in the literature (Bayer and Widen 2002, Mohan and Ramesh 2002, Pohl 1996b, Ramesh and Jarke 2001), and the semantics of the documents of our concern, we have identified nine different types of traceability relations between the various documents used in our approach. As shown in Table 4-2, the traceability relations are classified in six different groups.

Group 1: Relations between documents in the product line level and documents in the product member level (e.g. feature model vs. use case of PM1).

Group 2: Relations between documents of the same type for different product members (e.g. class diagram of PM1 vs. class diagram of PM2).

Group 3: Relations between documents of different types for the same product member (e.g. use case of PM1 vs. class diagram of PM1).

Group 4: Relations between documents of different types for different product members (e.g. use case of PM1 vs. class diagram of PM2).

Group 5: Relations between documents of the same type for the same product member (e.g. use case UC1 of PM1 vs. use case UC2 of PM1).

Group 6: Relations between different documents in the product line level (e.g. feature model vs. subsystem model).

Table 4- 2: Summary of traceability relation groups

	Feature Model	Subsystem Model	Process Model	Module Model	Use Case_1	Class Diagram_1	Sequence Diagram_1	Statechart Diagram_1	Use Case_2	Class Diagram_2	Sequence Diagram_2	Statechart Diagram_2
Feature Model		G6	G6	G6	G1	G1	G1	G1	G1	G1	G1	G1
Subsystem Model	G6		G6	G6	G1	G1	G1	G1	G1	G1	G1	G1
Process Model	G6	G6		G6	G1	G1	G1	G1	G1	G1	G1	G1
Module Model	G6	G6	G6		G1	G1	G1	G1	G1	G1	G1	G1
Use Case_1	G1	G1	G1	G1	G5	G3	G3	G3	G2	G4	G4	G4
Class Diagram_1	G1	G1	G1	G1	G3	G5	G3	G3	G4	G2	G4	G4
Sequence Diagram_1	G1	G1	G1	G1	G3	G3	G5	G3	G4	G4	G2	G4

	Feature Model	Subsystem Model	Process Model	Module Model	Use Case_1	Class Diagram_1	Sequence Diagram_1	Statechart Diagram_1	Use Case_2	Class Diagram_2	Sequence Diagram_2	Statechart Diagram_2
Statechart Diagram_1	G1	G1	G1	G1	G3	G3	G3	G5	G4	G4	G4	G2
Use Case_2	G1	G1	G1	G1	G2	G4	G4	G4	G5	G3	G3	G3
Class Diagram_2	G1	G1	G1	G1	G4	G2	G4	G4	G3	G5	G3	G3
Sequence Diagram_2	G1	G1	G1	G1	G4	G4	G2	G4	G3	G3	G5	G3
Statechart Diagram_2	G1	G1	G1	G1	G4	G4	G4	G2	G3	G3	G3	G5

Each of these groups can assist software development from different perspectives. For instance, relations in group 1 assist with the identification of reusable components; relations in group 2 and group 4 support comparisons between various product members in a product family; relations in group 3 and group 6 assist with better understanding of each product member and the core assets of product family, respectively; and relations in group 5 allow for the identification of evolution aspects in a product member and, therefore, supports the decision of when a new product member should be created. According to those groups, we have identified nine different types of traceability relations between the various documents described in the previous section. The nine types of traceability relations are not mutually exclusive. One or many types of traceability relations can exist between two particular artefacts. An example of each relation is given in figures 4-14 and 4-15. A description of each relation is given below.

Satisfiability: In this type of relation an element e1 *satisfies* an element e2, if e1 meets the expectation and needs of e2. A *satisfiability* relation may be arranged into group 1, group 3, group 4, or group 6, and hold between

(a) the description of a subsystem, process, or module model and the description of a feature in a feature model (group 6);

(b) the description of a subsystem, process, or module model and a feature in a feature model (group 6);

(c) an operation or attribute of a class in a class diagram and the description of a use case or the description of a feature in a feature model (group 1, group 3, or group 4);

(d) a transition in a statechart diagram and the description of a use case or the description of a feature in a feature model (group 1, group 3, or group 4);

(e) a sequence of events in a sequence diagram and the description of a use case or the description of a feature in a feature model (group 1, group 3, or group 4).

Dependency: In this type of relation an element e1 *depends on* an element e2, if the existence of e1 *relies on* the existence of e2, or if changes in e2 have to be reflected in

e1. A *dependency* relation may be arranged into group 1, group 3, group 4, or group 6, and hold between

(a) the description of a use case and the description of a feature in a feature model (group 1);

(b) an operation or attribute of a class in a class diagram and the description of a use case or a feature in a feature model (group 1, group 3, or group 4);

(c) a sequence of events in a sequence diagram and the description of a use case or a feature in a feature model (group 1, group 3, or group 4);

(d) a message in a sequence diagram and an operation of a class in a class diagram (group 3 or group 4);

(e) a transition in a statechart diagram and the description of a use case or a feature in a feature model (group 1, group 3, or group 4);

(f) a transition in a statechart diagram and a class in a class diagram (group 3 or group 4);

(g) a subsystem in a subsystem model, a process in a process model, or a module in a module model and a feature in a feature model (group 6);

(h) a class in a class diagram and a subsystem in a subsystem model, a process in a process model, or a module in a module model (group 1);

(i) a message in a sequence diagram and a transition in a statechart diagram, a message in a process model, or a message in a module model (group 1).

Overlap: In this type of relation an element e1 *overlaps* with an element e2, if e1 and e2 refer to common aspects of a system or its domain. This is a bi-directional relation. An *overlap* relation may be arranged into group 1, group 3, or group 4, and exists between

(a) the description of a feature in a feature model and a class in a class diagram, a state in a statechart diagram, or an object or message in a sequence diagram (group 1);

(b) the description of a use case and a class in a class diagram, a state in a statechart diagram, or an object or message in a sequence diagram (group 3 or group 4);

(c) a class in a class diagram and a state in a statechart diagram or an object in a sequence diagram (group 3 or group 4);

(d) an operation or attribute of a class in a class diagram and an operation or attribute of an object in a sequence diagram (group 3 or group 4);

(e) a state in a statechart diagram and a message in a sequence diagram (group 3 or group 4);

(f) a feature in a feature model and a use case (group 1);

(g) a feature in a feature model and a subsystem in a subsystem model, a process in a process model, or a module in a module model (group 1);

(h) the description of a process in a process model or the description of a subsystem in a subsystem model and a transition in a statechart diagram or a message in a sequence diagram (group 1);

(i) the description of a subsystem in a subsystem model, the description of a process in a process model, or the description of a module in a module model and a class in a class diagram (group 1).

Evolution: In this type of relation an element e1 *evolves to* an element e2, if e1 has been replaced by e2 during the development, maintenance, or evolution of the system. An *evolution* relation occurs between document models of the same type for the product member(s) in a family (group 5). This relation may hold between elements in

(a) use cases;

(b) class diagrams;

(c) statechart diagrams;

(d) sequence diagrams.

Implements: In this type of relation an element e1 *implements* an element e2, if e1 *executes* or *allows* for the achievement of e2. An *implements* relation may be arranged into group 1, group 3, or group 4, and hold between

(a) a class or an operation of a class in a class diagram and a feature in a feature model, flow of events in a use case, or the description of a use case (group 1, group 3, or group 4);

(b) a sequence of events in a sequence diagram and a feature in a feature model, flow of events in a use case, or the description of a use case (group1, group 3, or group 4);

(c) a transition in a statechart diagram and a feature in a feature model, flow of events in a use case, or the description of a use case (group 1, group 3, or group 4).

Refinement: This type of relation associates elements in different levels of abstractions. A refinement relation identifies how complex elements can be broken down into components and subsystems, and how elements can be specified in more details by other elements. Thus, an element e1 *refines* an element e2, when e1 specifies more details about e2. A *refinement* relation may be arranged into group 1, group 2, group 3, group 4, group 5, or group 6, and hold between

(a) the description of a subsystem in a subsystem model, the description of a process in a process model, or the description of a module in a module model and a feature in a feature model (group 6);

(b) a process model and a subsystem in a subsystem model (group 6);

(c) a module model and a process in a process model (group 6);

(d) a class in a class diagram, a sequence of events in a sequence diagram, or part of a statechart diagram and an event in a use case (group 2, group 3, group 4, or group 5);

(e) an object in a sequence diagram and a class in a class diagram (group 2, group 3, group 4, or group 5);

(f) a message in a sequence diagram and an operation of a class in a class diagram (group 2, group 3, group 4, or group5);

(g) a transaction and its corresponding source and target states in a statechart diagram and a message in a sequence diagram (group 2, group 3, group 4, or group 5);

(h) a class in a class diagram and a subsystem in a subsystem model (group 1);

(i) a sequence of events in a sequence diagram and a process in a process model or a module in a module model (group 1);

(j) a set of transitions in a statechart diagram and a process in a process model or a module in a module model (group 1);
(k) elements in different class diagrams, statechart diagrams, sequence diagrams, and use cases (group 3 or group 4).

Containment: In this type of relation an element e1 *contains* an element e2, when e1 is a document, or an element in a document, that uses an element e2, or a set of elements from a different document. This relation may be arranged into group 1, group 2, or group 5, and hold between

(a) a use case and a feature in a feature model (group 1);
(b) a subsystem in a subsystem model, a process in a process model, or a module in a module model and classes in a class diagram, when these elements contain the classes (group 1);
(c) a process model and a sequence of events in a sequence diagram or transitions in a statechart diagram (group 1);
(d) sequence or statechart diagrams and classes in a class diagram (group 2 or group 5).

Similar: This type of relation occurs between documents of the same type for different product members (group 2). This relation assists with the identification of common aspects between various product members. A *similar* relation is a bi-directional relation that may hold between elements in

(a) use cases;
(b) class diagrams;
(c) statechart diagrams;
(d) sequence diagrams.

A *similar* relation between elements e1 and e2 depends on the existence of a relation between e1 and another element e3 and a relation between e2 and element e3. For example, a use case uc1 is *similar* to a use case uc2, if both uc1 and uc2 hold a *containment* relation with a feature f1.

Similar relations between two elements can be derived from other relations based on the following inference rules:

(i) *Overlap relations*: if an element e1 overlaps an element e3, and an element e2 overlaps element e3, then element e1 is similar to element e2;

(ii) *Containment relations*: if an element e1 contains an element e3 and an element e2 contains element e3, then element e1 is similar to element e2;

(iii) *Satisfiability relations*: if an element e1 satisfies an element e3 and an element e2 satisfies element e3, then element e1 is similar to element e2;

(iv) *Refinement relations*: if an element e1 refines an element e3 and an element e2 refines element e3, then element e1 is similar to element e2;

(v) *Dependency relations*: if an element e1 depends on an element e3 and an element e2 depends on element e3, then element e1 is similar to element e2;

(vi) *Implements relations*: if an element e1 implements an element e3 and an element e2 implements element e3, then element e1 is similar to element e2.

Different: This type of relation also occurs between documents of the same type for different product members (group 2). This relation assists with the identification of variable aspects between various product members. More specifically, a different traceability relation expresses the different specialization of a particular variation point between two product members. A *different* relation is a bi-directional relation that may hold between elements in

(a) use cases;

(b) class diagrams;

(c) statechart diagrams;

(d) sequence diagrams.

A *different* relation between an element e1 and e2 depends on the existence of a relation between e1 and another element e3, and a relation between e2 and another element e4, where e3 and e4 are variants of the same variability point (e.g. subclasses of the same superclass, sibling features of the same parent feature). For example, a use case uc1 is *different* from a use case uc2, when there are two

subclasses c1 and c2 of the same parent class c, where c1 *implements* uc1 and c2 *implements* uc2. Consider the two use cases related to the *display of text message* (uc1) and *display of graphical message* (uc2) on mobile phones. Suppose a class diagram with class *Display_Screen* (c) with subclasses *Text_Screen* (c1) and *Graphic_Colour_Screen* (c2). The *Display_Screen* class has an operation *display*, which is inherited by classes c1 and c2. In this case, e1 implements uc1, e2 implements uc2, and uc1 and uc2 are different, although they have the same general purpose (*display of message*).

Different relations between two elements can be derived from other relations based on the following inference rules:

(i) Overlap relations: if an element e1 overlaps an element e3, an element e2 overlaps an element e4, and element e3 is a variant of element e4, then elements e1 and e2 are different;

(ii) Containment relation: if an element e1 contains an element e3, an element e2 contains an element e4, and element e3 is a variant of element e4, then elements e1 and e2 are different;

(iii) Satisfiability relations: if an element e1 satisfies an element e3, an element e2 satisfies an element e4, and element e3 is a variant of element e4, then elements e1 and e2 are different;

(iv) Implements relations: if an element e1 implements an element e3 and an element e2 implements an element e4, and element e3 is a variant of element e4, then elements e1 and e2 are different.

Figure 4-14 and Figure 4-15 show examples of each traceability relation being created between the examples of documents described in Section 4.2. Due to simplification, we only present the extract of each document. As shown in Figure 4-14, a *dependency* traceability relation holds between a subsystem *Messaging* in a subsystem model and a feature *Text Messages* in a feature model; a *satisfiability* traceability relation holds between the description of a module *Short Messaging* in a module model and a feature *Text Messages* in a feature model; a *refinement* traceability relation holds between a process model *P1* and a subsystem *Messaging* in a subsystem model; two *containment* traceability relations hold between use cases (*UC1* and *UC2*)

and a feature *Text Messages* in a feature model; and a *similar* traceability relation holds between two use cases *UC1 and UC2* and based on *containment* traceability relations existing between use cases *UC1* and *UC2* and a feature *Text Messages* in a feature model.

As shown in Figure 4-15, an *overlap* traceability relation holds between an operation *takingPhoto* of a class *Camera* in a class diagram and an operation *takingPhoto* of an object *Camera* in a sequence diagram; an *evolution* traceability relation holds between two statechart diagrams (*SD1 and SD2*); two *implement* traceability relations hold (i) between an operation *takingPhoto* of a class *Camera* and the description of a use case *UC3,* and (ii) between an operation *takingPhoto* of a class *CameraZoom2x* in a class diagram and the description of a use case *UC4*; and a *different* relation holds between two use cases *UC3 and UC4*. Examples of traceability rules to identify these traceability relations in Figure 4-14 and Figure 4-15 will be described in Chapter 5.

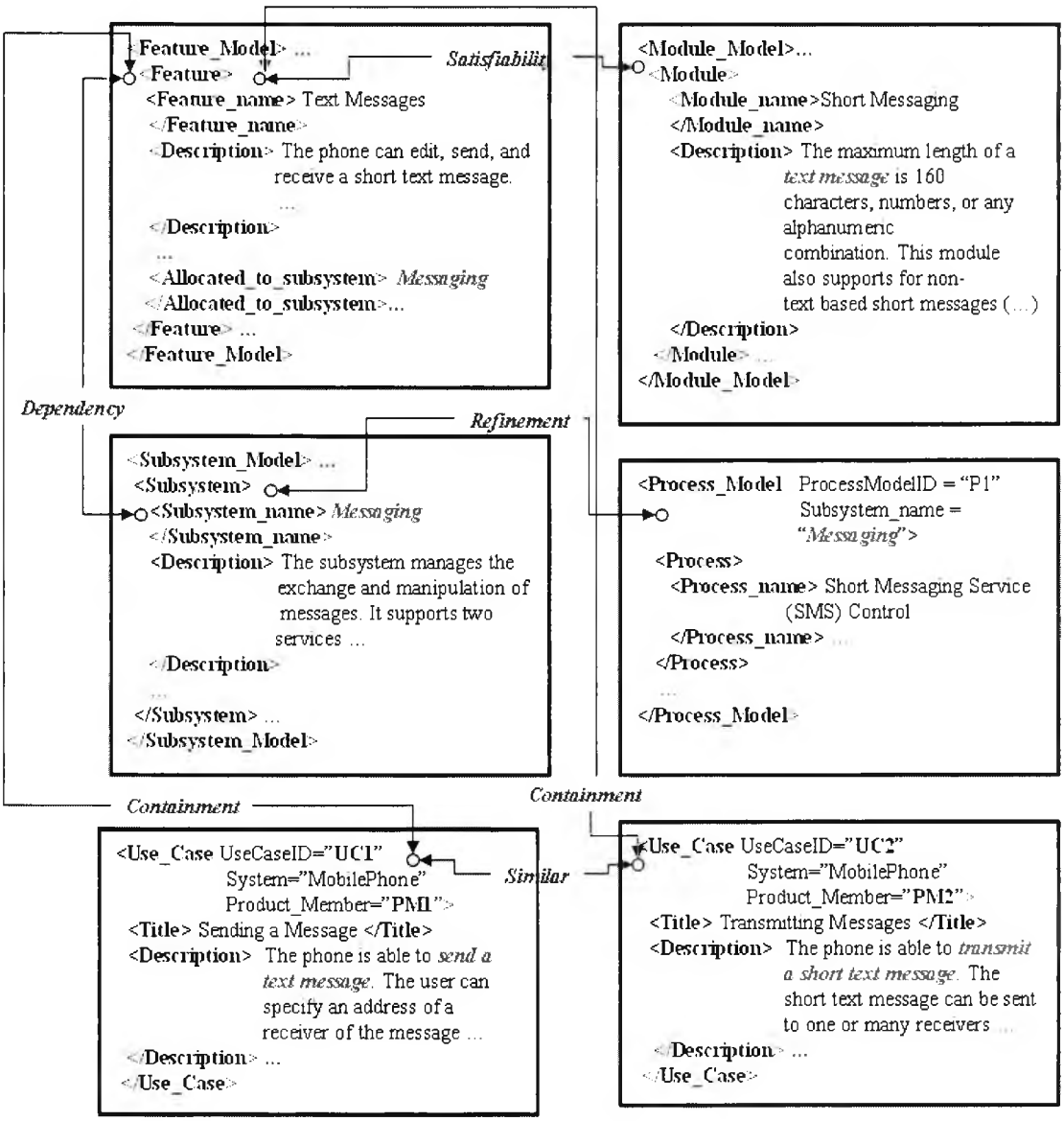


Figure 4- 14: Examples of *satisfiability*, *dependency*, *refinement*, *containment*, and *similar* traceability relations

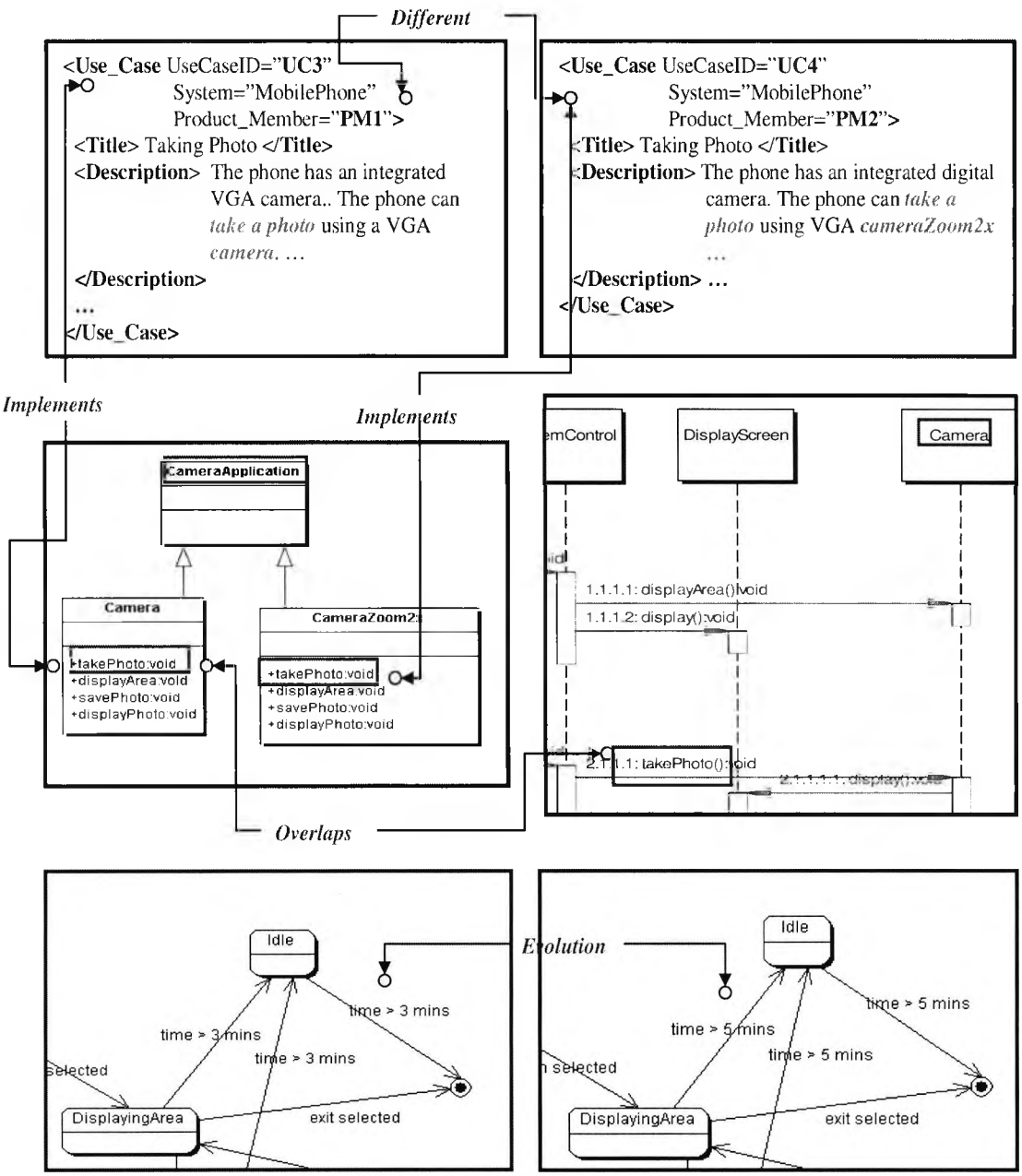


Figure 4- 15: Examples of *implements*, *overlaps*, *evolution*, and *different* traceability relations

Table 4-3 presents a summary of our traceability reference model. In the table, each cell contains the different types of traceability relations that may exist between the artefacts described in the row and column of that cell. In the table we do not represent the exact elements that are related in the different artefacts, but represent the types of the artefacts. The direction of the relation is represented from a row *[i]* to a column *[j]*. Thus, a relation type *rel_type* in a cell *[i][j]* signifies that "*[i] is related to [j] though rel_type*" (e.g. "subsystem model *satisfies* feature model"). The traceability relations that are bi-directional appear in the two correspondent cells for that relation.

Table 4- 3: Traceability Reference Model

	Feature Model	Subsystem Model	Process Model	Module Model	Use Case	Class Diagram	Statechart Diagram	Sequence Diagram
Feature Model		*Overlap*	*Overlap*	*Overlap*		*Overlaps*	*Overlaps*	*Overlaps*
Subsystem Model	*Satisfies* *Depends_on* *Refines* *Overlap*					*Contains*		
Process Model	*Satisfies* *Depends_on* *Refines* *Overlap*	*Refines*				*Contains*	*Contains*	*Contains*
Module Model	*Satisfies* *Depends_on* *Refines* *Overlap*		*Refines*			*Contains*		
Use Case	*Contains* *Depends_on*				*Similar* *Different* *Evolves*	*Overlaps*	*Overlaps*	*Overlaps*
Class Diagram	*Satisfies* *Depends_on* *Overlaps* *Implements*	*Refines* *Depends_on*	*Refines* *Depends_on*	*Refines* *Depends_on*	*Satisfies* *Depends_on* *Overlaps* *Implements* *Refines*	*Similar* *Different* *Evolves*	*Overlaps*	*Overlaps*
Statechart Diagram	*Satisfies* *Depends_on* *Overlaps* *Implements*		*Refines* *Depends_on*	*Refines* *Depends_on*	*Satisfies* *Depends_on* *Overlaps* *Implements* *Refines*	*Depends_on* *Overlaps* *Contains*	*Similar* *Different* *Evolves*	*Overlaps* *Refines*
Sequence Diagram	*Satisfies* *Depends_on* *Overlaps* *Implements*		*Refines* *Depends_on*	*Refines* *Depends_on*	*Satisfies* *Depends_on* *Overlaps* *Implements* *Refines*	*Depends_on* *Overlaps* *Refines* *Contains*	*Overlaps*	*Similar* *Different* *Evolves*

4.4. Summary

This chapter described a traceability reference model for product family systems. It has presented the software artefacts used in our approach and the different types of traceability relations that exist between these artefacts. In the next chapter, we describe our approach to allow automatic generation of those traceability relations

Chapter 5

Traceability Framework

This chapter elaborates the approach on how to establish the traceability relations on the domain of product family systems. We present the framework of traceability to the software product family systems according to the traceability reference model described in previous chapter. This chapter illustrates the main process, methodology and techniques of our approach. Section 5.1 gives an overview of the traceability generation process. Section 5.2 describes traceability rules, traceability relations, and examples of direct and indirect traceability rules. Section 5.3 describes the extended functions implemented in XQuery and Java. Section 5.4 summarises the chapter.

5.1. Overview of The Traceability Generation Process

Our approach is based on the extensible markup language (XML) technology since there are several reasons:

(a) XML has become the de facto language to support data interchange among heterogeneous tools and applications;

(b) the existence of large number of applications that use XML to represent information internally or as a standard export format (e.g. Unisys XML exporter for Rational Rose (IBM), Borland Together (Borland), ArgoUML (ArgoUML) , and

(c) XML allows the use of XQuery (W3C) as a standard way of expressing traceability rules.

The XML documents used in our approach are based on XML schemas. We have created XML schemas for the feature models, subsystem models, process models, module models, and use cases. These XML schemas are described in Appendix A. Our work also adopts XQuery (W3C) as a rule representation language due to several reasons:

(a) XQuery is powerful for retrieving data in XML documents;
(b) XQuery is compatible with XML and Java environment; and
(c) XQuery is extensible and flexible and, therefore, it enables extensions of the language, when necessary.

Apart from the embedded functions offered by XQuery, it is possible to add new functions and commands. We have extended XQuery to support the representation of consequence part of the rules, i.e. the actions to be taken when the conditions are satisfied, and to support extra functions to cover some of the traceability relations. More specifically, these functions have been implemented in XQuery and java and are concerned with the identification of specific elements in the documents and words that are synonyms, or textual comparison. These functions are explained in more details in Section 5.3. Our traceability framework focuses on the generation of traceability between the software artefacts in the domain of product family systems. Figure 5-1 presents an overview of the traceability generation process, which is composed of three main stages, namely:

(a) Annotation of textual sentences in the documents with part-of-speech (POS) assignments (*Grammatical Tagging*), using CLAWS C7 (CLAWS). The documents that mainly contain textual sentences (i.e. use cases, feature model, subsystem model, process model, and module model) are annotated with POS tags. The POS tags generated by CLAWS are converted into XML tags are shown in the examples in Figures 4-3, 4-4, 4-6, 4-8, and 4-10. Our approach can also support tagging of diagram elements (e.g. subsystem, process, and module model names).

(b) Creation of documents in XML format (*XML Creation*), based on the XML schemas and the POS tags generated by CLAWS, or based on an XMI format (XMI), as described in Chapter4.

(c) Generation of direct and indirect traceability relations (*Traceability Generation*), based on traceability rules and extra functions.

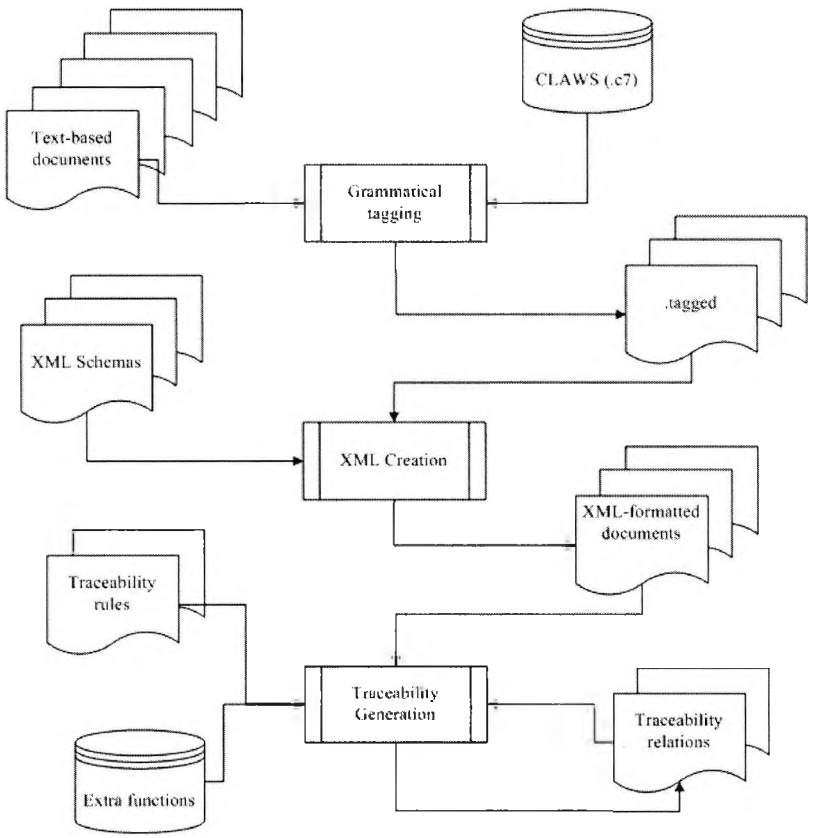


Figure 5- 1: Overview of traceability generation process

The traceability generation process is illustrated in Figure 5-2. More specifically, traceability relations are generated by a *Traceability Generator* component that we have developed with is formed by two sub-components.

(a) *Rule inference* sub-component is responsible for:

- identifying the traceability rules that are related to different types of documents to be traced and different types of traceability relations to be generated, and
- instantiating the placeholders for the document types in the identified rules with the names of the documents to be traced (see Section 5.2). The information about the traceability documents to be traced and traceability relations to be generated are given by the user (see Chapter 6).

(b) The *rule parser* sub-component is responsible for executing the identified and instantiated rules. It uses the XML-formatted documents, extra functions, and WordNet 2.0 (WordNet) to assist with the identification of synonyms. The direct and indirect traceability relations resulting from the execution of the rules are represented in XML documents (Direct_Trace_Rel.xml and Indirect_Trace_Rel.xml, respectively). The document with direct traceability relations is used as input to the rule parser to generate indirect traceability relations.

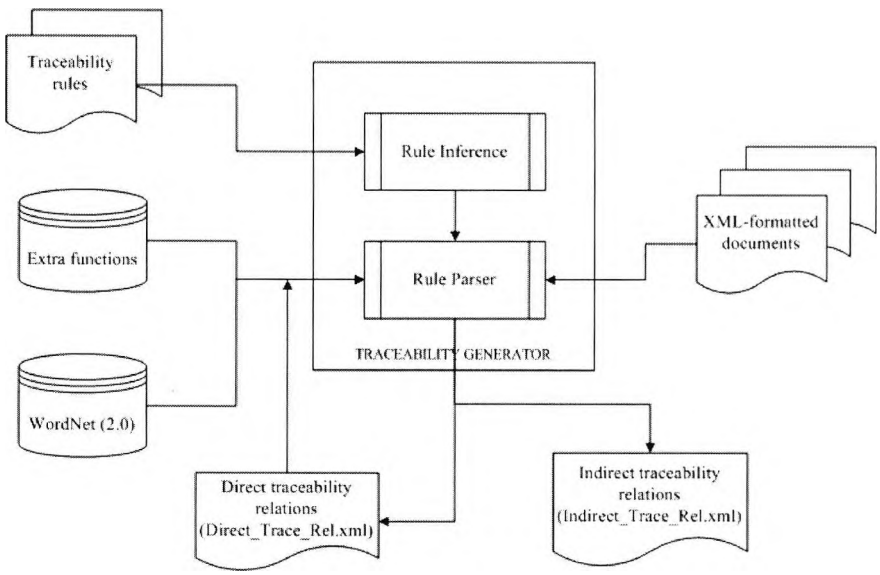


Figure 5- 2: ***Traceability generator***

The traceability generation process can fit with some existing methods for product family system development according to the current literature. Examples of those methods are such as FODA, FORM, and FeatuRSEB.

Additionally, the traceability rules used by *traceability generator* have been created based on:

(i) semantic of the documents being compared;

(ii) the various types of traceability relations in the product family domain;

(iii) the grammatical roles of the words in the textual parts of the documents; and

(iv) synonyms and distance of words being compared in a text.

Case (i): As an example of case (i), a rule for comparing feature and use case models take into consideration the fact that a feature model specifies requirements at the product line level, while use cases describe requirements for product members which may be more specific. Therefore, it may be necessary to traverse the hierarchy of a feature and investigate if one or more children of a feature appear in the use case. Similarly, a sequence diagram describes the order in which messages are exchanged between various class objects and, therefore, rules for comparing operations in classes and sequence of messages in a sequence diagram should be used.

Case (ii): Regarding case (ii), the types of traceability relations also play an important role in the various traceability rules. It is not necessary to create traceability rules for identifying evolution relations between elements in feature models and class diagram, feature models and sequence diagram, or feature models and statechart diagrams since such relations do not exist between the above documents (evolution relations exist between documents of the same type for the same product member).

Case (iii): Considering case (iii), it is a common approach that names given by software engineers for the main elements in class, sequence, and statechart diagrams do not contain certain types of words such as articles, coordinating and

subordinating conjunctions, singular and plural determiner, comparative and superlative adjectives, etc. Therefore, when comparing e.g. descriptions of use cases and feature names, or flow of events in use cases with elements in the above diagrams (e.g. classes, messages, operations, and transitions), the above types of words do not need to be considered.

More specifically, words are not concerned when they are annotated with POS tags such as articles (e.g. the, no, a, an, every), conjunctions (e.g. and, or, but, because), determiners (e.g. whose, these, which, this, that, any, some, all, half, little, much, few, several, many, such), adjectives (e.g. old, able to, willing to), pronouns (e.g. he, his, him), adverbs (e.g. more, less, however, about, when, where, now, tomorrow), and interjections (e.g. oh, yes, um). In the other words, the words annotated with POS tags and categorized as *verb* or *noun* in texts are considered. Examples of POS tags categorized as noun are NN1, which is a singular common noun (e.g. book, girl), NN2, which is a plural common noun (e.g. books, girls), and examples of POS tags categorized as verb are VV0, which is a base form of lexical verb (e.g. give, work), VVD, which is past tense of lexical verb (e.g. gave, worked), VVG, which is verb-ing participle of lexical verb (e.g. giving, working), VVI, which is infinitive (e.g. to give, to work), VVN, which is past participle of lexical verb (e.g. given, worked), and VVZ, which is -s form of lexical verb (e.g. gives, works).

Case (iv): With respect to case (iv), the multiplicity of stakeholders participating in the development of the system, the different phases of software product line engineering (domain analysis vs. domain design), and the different level of specialization of the system (product line vs. product members) may lead to the use of different words to represent the same thing. More specifically, our approach supports the use of equivalent words to specify documents. As described, our work has applied WordNet 2.0 as the database of synonyms and Java API, *Java WordNet Library* (JWNL), to access the WordNet database.

Furthermore, the existence of two or more words in a paragraph description does not imply that the paragraph is related to these words, in particular when the words

appear in different sentences in the paragraph or in different phrases in the same sentence. As an example consider part of a paragraph describing some functionalities of a mobile phone in a use case as shown below, and the operation *receive_call()* in class *Phone*. In this case, although the paragraph contains the words "receive" and "call", the text in the paragraph is not concerned with the "receive of calls", but with "receive of textual messages" and the time allowed for international calls. If the distances of the words in the paragraph were not taken into consideration, the operation would have been incorrectly related to the description of the use case.

<Description> The phone should be able to send and **receive** textual messages. (...)
An international phone **call** should not last more than 10 minutes.
(...)
<\Description>

We describe below traceability rules being used and traceability relations being created by *traceability generator*.

5.2. Traceability Rules and Relations

In our approach, the generation of traceability relations is based on the use of traceability rules due to some reasons:

(a) it enables automatic traceability;

(b) it allows for a standard way of representing criteria for identifying traceability relations i.e. equivalent words from different text, appropriate distance of words in a text, grammatical roles of words in a text, and various types of traceability relations;

(c) it supports the processing of large-sized documents and the creation of a large number of traceability relations; and

(d) it supports consideration into the semantics of documents which express the interdependencies between a product family.

As below, we describe the template of traceability rules being used in our approach.

Traceability Rules

We use two different types of traceability rules, namely (a) *direct traceability rules* and (b) *indirect traceability rules*. Type (a) is concerned with traceability rules for direct relations between two independent elements such as satisfiability, dependency, overlaps, evolution, implements, refinement, and containment relations; while type (b) is concerned with traceability rules for relations that depend on the existence of other relations such as similar and different relations.

```
TRACE_RULE    RuleID     = R_ID
              RuleType   = Rule_Type
              DocType1 = DocTypeName
              DocType2 = DocTypeName
    QUERY
      [DECLARE Namespace]
      [DECLARE Functions]
      [DECLARE Variables]
      for $variable_name1 in doc(DocType1Placeholder)//XPathExpression
          $variable_name2 in doc(DocType1Placeholder)//XPathExpression
      where
          fi(fi+1...(fi+j(•))...)
    QUERY_END
    ACTION
      RELATION   RuleID     = R_ID
                 Type       = Relation_Type
                 DocType1 = DocTypeName
                 DocType2 = DocTypeName
      ELEMENT    Document = DocName    [ElementType1] $variable_name1
                                       [/XpathExpression]
                                       [ElementType2]
      ELEMENT  Document = DocName      [ElementType1] $variable_name2
                                       [/XpathExpression]
                                       [ElementType2]
      [RelationType {XpathExpression} {XpathExpression}]
      [RelationType {XpathExpression} {XpathExpression}]
  ACTION_END
TRACE_RULE_END
```

Figure 5- 3: Traceability rule template

Figure 5-3 shows a general template for *direct* and *indirect* traceability rules. In the template, elements between square brackets ("[" ,"]") are optional, and fi(fi+1...(fi+j(•))...) are embedded XQuery functions or the extra functions that

we have developed. The XML Schema for our traceability rules can be found in Appendix A. Both types of traceability rules are composed of three main parts, as described below. An example of a traceability rule for a *containment* traceability relation between use cases and feature models is shown in Figure 5-4 and an example for *similar* relation is shown in Figure 5-5. We explain below the different parts in a traceability rule.

Part 1: It consists of the rule identification and contains a unique *RuleID*, description of the type of the rule (*RuleType*), and descriptions of the types of documents associated with the rule (*DocType1*, *DocType2*). The rule type is based on the type of traceability relation generated by the rule. In the case of direct traceability rules, attributes *DocType1* and *DocType2* contain the names of the different types of documents used in our approach (Figure 12); while for indirect traceability rules, attributes *DocType1* and *DocType2* refer to the XML_Base_Relationship document that contains the results of previously identified relations (Figure 5-5).

Part 2: It is represented by element <Query> and consists of XQuery statements. This part is composed of three other subparts.

The first subpart (*declare*) is optional and contains declaration of namespaces, variables, or extra functions used in the rule. In our approach, the extra functions that we have developed are either implemented as XQuery statements (viz. XQuery_functions) or as Java classes (viz. Java_functions). The XQuery_functions are declared as *function*. The Java_functions are represented as Java packages and declared as *namespace*. Figure 5-4 shows an example of these declarations for Java functions. The example in Figure 5-5 does not make use of any declaration.

The second subpart (*for*) identifies elements of the documents (*DocType1* and *DocType2*) to be compared and bind these elements to variables *$item1* and *$item2*, respectively (*$variable_name1* and *$variable_name2* in Figure 5-3). Initially, the elements to be compared are described in XPath (XPath) expressions associated

with placeholders that represent the types of documents to be traced. The placeholders for the documents to be traced are automatically substituted by specific document names (file names) after the user has indicated these documents by using the traceability tool (see Chapter 6). The examples in Figure 5-4 and Figure 5-5 show the values for *$item1* and *$item2* already instantiated with the document names (UseCase_UC1.xml and Feature_MP.xml in Figure 5-4 with the XPath expressions for the respective elements, and Direct_Trace_Rel.xml in Figure 5-5 with XPath expressions for relations of type *containment*). In the case of indirect traceability rules $item1 and $item2 always refer to Direct_Trace_Rel.xml. However, the type of the relation given by the XPath expression (\\Relation[@type=" "]) differ depending on the rule type.

The third subpart (*where*), describes the *condition* part of the rule that should be satisfied in order to create a traceability relation. The condition part can use a sequence, conjunction, or disjunction of XQuery in-built functions (e.g. *some*, *contains*, *satisfies*) or the extra XQuery or Java functions that we have implemented. Depending on the rule, the condition part also takes into consideration the XML POS-tags in the textual parts of the documents.

In the example of Figure 5-4, the rule verifies if the words (or their set of synonyms) in element *Title* of UseCase_UC1 appear in the *Description* of a feature in Feature_MP.xml, at an appropriate distance (Java class *checkDistanceControl*). The rule checks for synonyms, by using WordNet (WordNet), of any possible form of the main verb (VVI, VVB, VVG, VV0) and of any possible form of the noun (NN0, NN1, NN2, NP0) of the verb-phrase in the title of the use case. In Figure 5-5, the rule verifies if there are two relations of type *containment* in Direct_Trace_Rel.xml document between a use case and a feature model such that the feature names are the same (Element[2]) and the use cases are different (Element[1]).

Part 3: It is represented by element (<Action>) and describes the *consequence* part of the rule. It specifies the action(s) to be taken if the conditions in Part 2 are satisfied. The consequence part describes the type of traceability relation to be created

(attribute Type) and the elements that should be related through it in the documents described in the *for* part of the rule (element <Element>). For the case of direct traceability rules, an extra element associated with each element (ElementType2 in Figure 5-3) may be used to indicate the exact type of elements in the respective documents that were satisfied by the rule, when necessary. The extra element represented by ElementType1 in Figure 5-3 is used when the content of *$variable_name1* or *$variable_name2* is of type string and it is necessary to represent the XML element that this content represents (see rules R4 and R6 in Figure 5-12 and Figure 5-14 for examples). For the case of indirect traceability rules, a special element is used to represent how the elements being compared are similar or different (RelationType in Figure 11). The content of element <Action> is used to compose the *return* part of XQuery. The implementation of an action consists of writing the information in the <Action> part, in the XML relation document (Direct_Trace_Rel.xml and Indirect_Trace_Rel.xml). As in Part 2, the placeholders of the specific document models to be associated are instantiated based on the user's input

```
<TraceRule RuleID="R1" RuleType="containment"
            DocType1="Use Case" DocType2="Feature Model">
 <Query>
  declare namespace s="java:synonym.s";
  declare namespace d="java:distanceControl.d";

  for $item1 in doc("file:///c:/UseCase_UC1.xml")//Use_Case,
      $item2 in doc("file:///c:/Feature_MP.xml")//Feature_Model/Feature
  where
    d:checkDistanceControl($item2/Description,
    s:setof(s:findSynonym($item1/Title/VVI),s:findSynonym($item1/title/VVB),
          s:findSynonym($item1/Title/VV0), s:findSynonym($item1/Title/VVG)) ,
    s:setof(s:findSynonym($item1/Title/NN0), s:findSynonym($item1/Title/NN1),
          s:findSynonym($item1/Title/NP0),s:findSynonym($item1/Title/NN2)))
 </Query>
 <Action>
    <Relation RuleID="R1" Type="containment"
              DocType1="Use Case" DocType2="Feature Model">
    <Element Document="file:///c:/UseCase_UC1.xml"> {$item1/Title} </Element>
    <Element Document="file:///c:/Feature_MP.xml"> {$item2/Feature_name} <Description/>
    </Element>
    </Relation>
 </Action>
</TraceRule>
```

Figure 5- 4: Example of *containment* traceability rule

In Figure 5-4 a relation of type *containment* is created between the title of use case UseCase_UC1 (first <Element>) and the feature name in Feature_MP.xml document that satisfies the condition part of the rule (second <Element>) represented by XPath expressions. An element <Description> is used to indicate that the relation is between the title of the use case and the description of the feature. In Figure 5-5, a relation of type *similar* is created between the titles of the two use cases (both elements <Element>) together with an extra element representing how the two use cases are similar, i.e. through a *containment* relation with the feature (element <Containment>).

```
<TraceRule RulID="R2" RuleType="similar"
          DocType1="XML-Based-Rel" DocType2="XML-Based-Rel">
  <Query>
   for  $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="containment"],
          $item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="containment"]
     where
    $item1/@DocType1="Use Case" and $item1/@DocType2="Feature Model" and
    $item2/@DocType1="Use Case" and $item2/@DocType2="Feature Model" and
    string($item1/Element[2]) =  string($item2/Element[2]) and
    $item1/Element[1]/@Document != $item2/Element[1]/@Document

  </Query>
  <Action>
    <Relation RuleID="R2" Type = "similar" Term ="use case contains feature model">
        <Element>{$item1/Element[1]/@Document} {$item1/Element[1]/Title} </Element>
        <Element>{$item2/Element[1]/@Document} {$item2/Element[1]/Title} </Element>
        <Containment>{$item1/Element[2]/@Document}{$item1/Element[2]   /Feature_name}
        </Containment>
    </Relation>
  </Action>
</TraceRule>
```

Figure 5- 5: Example of *similar* traceability rule

Traceability Relations

An example of rule R1 in Figure 5-5 exists between use case UC1 of product member PM_1 entitled *Sending a Message* (Figure 4-4) and feature named *Text Messages* (Figure 4-3). A *containment* relation is created since a synonym (send) of verb <VVG> Sending </VVG> and noun <NN1> Message </NN1> appear in

the description of the feature at an appropriate distance; i.e., a sequence of a conjunction of verbs (<VVI> send </VVI> <SC>,</SC>, <CJC>and</CJC>, <VVI> receive</VVI>), followed by a qualifier of the noun message (<AT0> a</AT0> <AJ0>short</AJ0> <NN1> text </NN1>), separate the words *send* and *message*. The result of rule R1 for use case UC1 and feature *Text Messages* are shown in Figure 5-6.

```
<Relation_Document>
  <Relation RuleID="R1" Type="containment"
            DocType1="Use Case" DocType2="Feature Model">
    <Element Document="file:///c:/UseCase_UC1.xml">
        <Title> <VVG> Sending </VVG> <AT0> a </AT0> <NN1> Message </NN1>
        </Title>
    </Element>
    <Element Document="file:///c:/Feature_MP.xml">
        <Feature_name> <NN1> Text </NN1> <NN2> Messages </NN2>
            <Description></Description>
        </Feature_name>
    </Element>
  </Relation>
  <Relation RuleID="R1" Type="containment"
            DocType1="Use Case" DocType2="Feature Model">
      <Element Document="file:///c:/UseCase_UC2.xml">
        <Title> <VVG> Transmitting </VVG> <NN2> Messages </NN2> </Title>
      </Element>
      <Element Document="file:///c:/Feature_MP.xml">
          <Feature_name> <NN1> Text </NN1> <NN2> Messages </NN2>
            <Description></Description>
        </Feature_name>
      </Element>
  </Relation>
  ...
</Relation_Document>
```

Figure 5- 6: Result of *containment* traceability relations

Another example of rule R1 also exists between use case UC2 of product member PM_2 entitled *Transmitting Messages* in Figure 5-7 and feature *Text Messages* in Figure 4-3. In this case, a *containment* relation is also created, as shown in Figure 5-6.

```
<Use_case UseCaseID="UC2" System="MobilePhone" Product_Member="PM2">
  <Title> <VVG> Transmitting </VVG> <NN1> Message </NN1> </Title>
  <Description>
    ...<NN1>phone</NN1>  <VM0 >can</VM0><VVI>send</VVI>...
  </Description>
  ...

</Use_case>
```

Figure 5- 7: Example of use case UC 2 *Transmitting Message*

As shown in Figure 5-6, the use cases UC1 *Sending a Message* and UC2 *Transmitting Messages* have *containment* traceability relations with the same feature *Text Messages* (Figure 4-3). In this case, a *similar* relation is created by the deployment of the rule R2 (Figure 5-5) as shown in Figure 5-8.

```
<Relation RuleID = "R2" Type ="similar" Term = "use case encompass feature model">
    <Element Document="file:///c:/UseCase_UC1.xml">
        <Title>  <VVG> Sending </VVG> <AT0> a </AT0> <NN1> Message </NN1>
        </Title>
    </Element>
    <Element Document="file:///c:/UseCase_UC2.xml">
        <Title>  <VVG> Transmitting </VVG> <NN2> Messages </NN2> </Title>
    </Element>
       <Containment Document="file:///c:/Feature_MP.xml">
            <Feature_name> <NN1>Text</NN1> <NN2>Messages</NN2>
            </Feature_name>
      </Containment>
</Relation>
```

Figure 5- 8: Result of similar traceability relation

Our approach use XML for representing traceability relations due to some reasons:

(a) It guarantees the representation of large number of relations that are automatically created by traceability generator;

(b) It allows the maintenance and visualization of traceability relations by using common commercial XML-based applications e.g. XML-spy, TurboXML, or general-purposed applications e.g. web browsers, text editors; and

(c) It allows the retrieval of traceability relations by using XQuery statements.

More specifically, all traceability relations are represented in XML documents. The direct relations and indirect relations are separately recorded in "Direct_TraceRel.xml" and "Indirect_TraceRel.xml", respectively. Additionally, the logs of the traceability activity are recorded in XML documents ("Instantiated_Direct_TraceRule.xml" and "Instantiated_Indirect_TraceRule.xml" for direct and indirect relations respectively).

Examples of Direct and Indirect Traceability Rules

Apart from the examples of traceability rules for *containment* and *similar* relations shown in Figures 5-4 and 5-5, in this subsection we present examples of traceability rules for each of the other traceability relation types used in our work. These traceability rules follow from the examples shown in Figure 4-14 and Figure 4-15. We repeat these figures below as Figure 5-9 and Figure 5-10, respectively, to facilitate understanding.

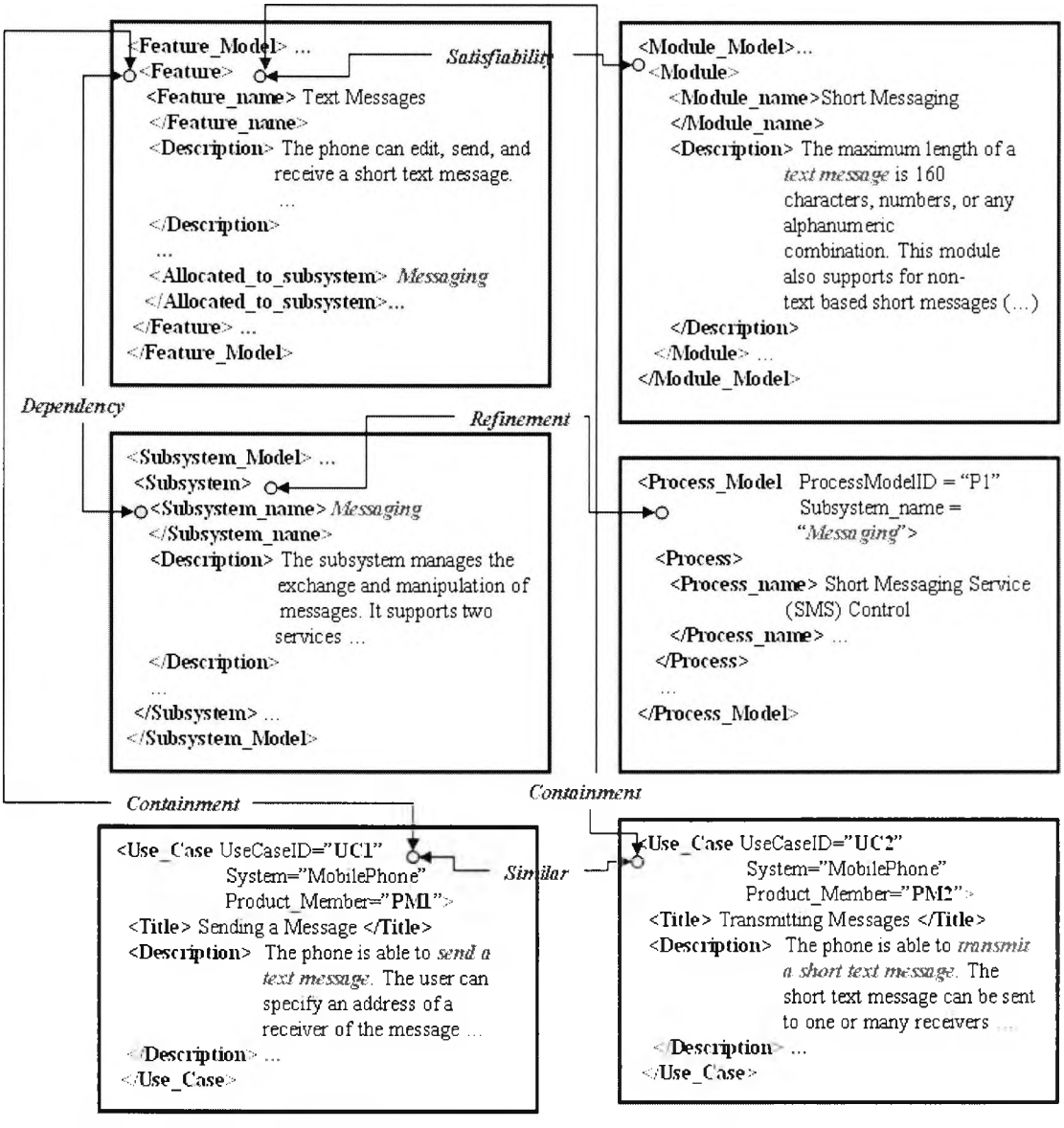


Figure 5- 9: Examples of traceability relations (repetitive to Figure 4-14)

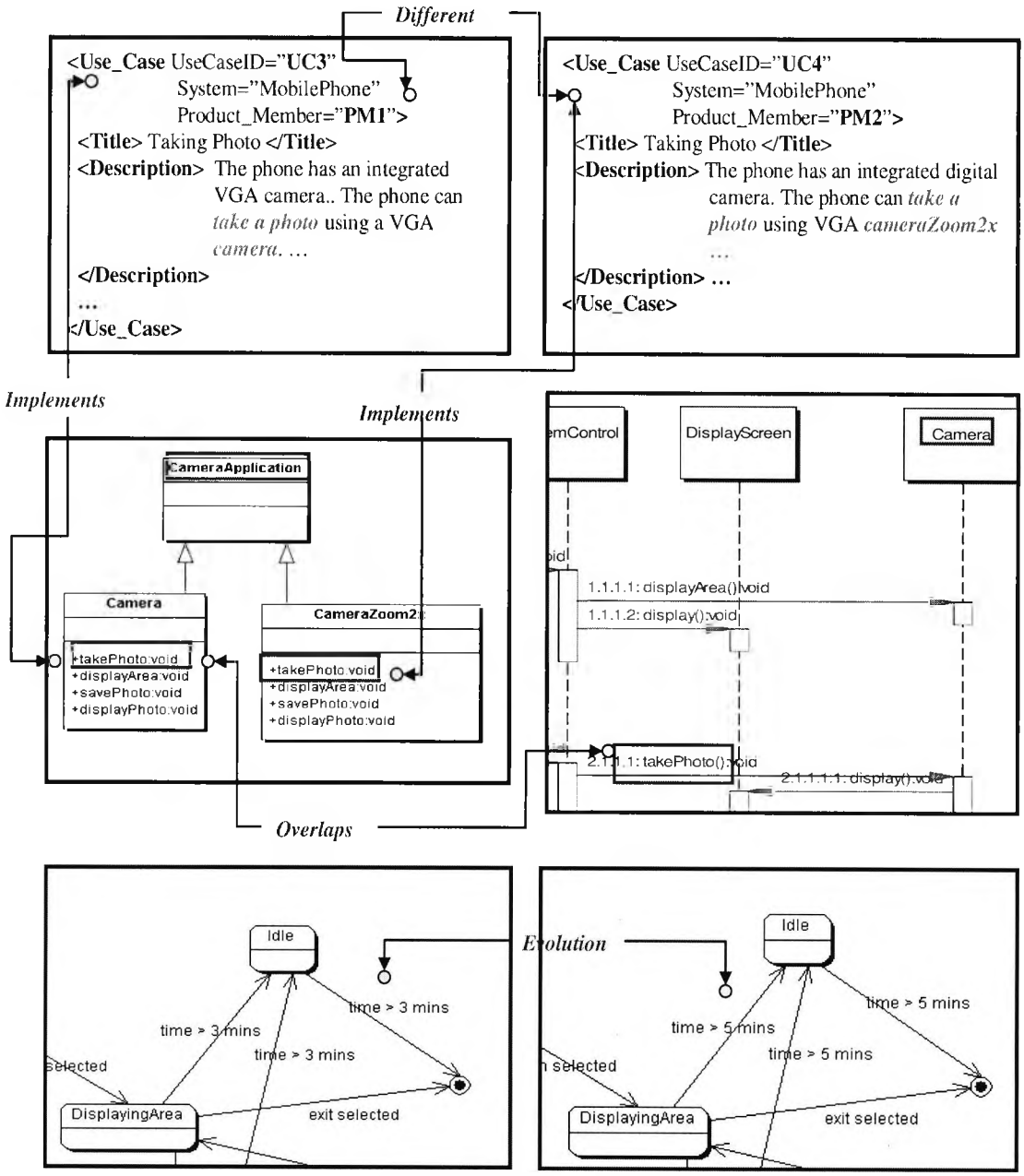


Figure 5- 10: Examples of traceability relations (repetitive to Figure 4-15)

Rule R3: Dependency

As shown in Figure 5-11, this rule can establish a *dependency* relation between a subsystem in a subsystem model and a feature in a feature model when the name of the subsystem is the same as the content of element <Allocated_to_Subsystem> in a feature model. In this case, the feature has been allocated to the subsystem and

any change in the feature has to be reflected in the subsystem. In this rule an XQuery function (*normalize-space()*) is used to remove any XML sub-elements and white spaces composing the content of elements <Subsystem_name> and <Allocated_to_Subsystem>. According to Figure 5-9, a *dependency* relation exists between subsystem *Messaging* and feature *Text Messages*, since this feature is allocated to subsystem *Messaging*. The relation is created between the name of subsystem in *SubsystemModel.xml* (first <Element>) and the feature name in *Feature_MP.xml* (second <Element>).

```
<TraceRule RuleID="R3"  RuleType="dependency"
           DocType1="Subsystem Model" DocType2="Feature Model">
  <Query>

       for  $item1 in doc("file://c:/SubsystemModel.xml")//Subsystem/Subsystem_name,
            $item2 in doc("file://c://Feature_MP.xml")//Feature_Model/Feature
                                                        /Allocated_to_Subsystem
       where
            normalize-space($item1) = normalize-space($item2)
  </Query>
  <Action>
    <Relation RuleID="R3" Type="dependency"
              DocType1="Subsystem Model" DocType2="Feature Model">
      <Element Document="file://c:/SubsystemModel.xml"> {$item1} </Element>
      <Element Document="file://c://Feature_MP.xml"> {$item2/../Feature_name}
                </Element>
    </Relation>
  </Action>
</TraceRule>
```

Figure 5- 11: Example of *dependency* traceability rule

Rule R4: Refinement

As shown in Figure 5-12, this rule can establish a *refinement* relation between a process model and a subsystem when the content of the attribute *Subsystem_name* of the process is the same as the name of the subsystem. This rule uses the Java function *stringnospace()* to compare the names of the elements without white spaces. An example of a *refinement* traceability relation from this rule exists between subsystem *Messaging* and process model *P1* as shown in Figure 5-9. The relation is created between the identifier of the process model *ProcessModel.xml* represented in <ProcessModelID> (in the first <Element>) and the name of subsystem in *SubsystemModel.xml* (second <Element>).

```
<TraceRule RuleID="R4"  RuleType="refinement"
           DocType1="Process Model" DocType2="Subsystem Model">
  <Query>
    declare namespace d="java:distanceControl.d";

    for $item1 in doc("file://c:/ProcessModel.xml")//Process_Model
        $item2 in doc("file://c:/SubsystemModel.xml")//Subsystem_Model/Subsystem
                                                          /Subsystem_name

    where
        d:stringnospace($item1/@Subsystem_name) = d:stringnospace( $item2)
  </Query>
  <Action>
    <Relation RuleID="R4" Type="refinement"
              DocType1="Process Model" DocType2="Subsystem Model">
      <Element Document="file://c:/ProcessModel.xml">
          <ProcessModelID> {$item1/@ProcessModelID} </ProcessModelID>
                                                          </Element>
      <Element Document="file://c:/SubsystemModel.xml">{$item2 } </Element>
    </Relation>
  </Action>
</TraceRule>
```

Figure 5- 12: Example of *refinement* traceability rule

Rule R5: Satisfiability

As shown in Figure 5-13, this rule can establish a *satisfiability* relation between the description of a module model and a feature when the name of a feature appears in the description of a module model at an appropriate distance. This rule uses *constainsInDistance()* extra function to verify if the name of the feature, or any of its synonyms, is in the description of the module. A variation of this rule takes into consideration the name of a feature and the name of its parent feature when such parent exists. An example of a *satisfiability* traceability relation from this rule is shown in Figure 5-9 between feature *Text Messages* and the description of module named *Short Messaging*. The relation is created between the name of module in *ModuleModel.xml* (first <Element>) and the feature name in *Feature_MP.xml* (second <Element>).

```
<TraceRule RuleID="R5"  RuleType="satisfiability"
            DocType1="Module Model" DocType2="Feature Model">
  <Query>
     declare namespace d="java:distanceControl.d";

     for  $item1 in doc("file://c:/ModuleModel.xml")//Module_Model/Module/Description,
          $item2 in doc("file://c:/Feature_MP.xml")//Feature_Model/Feature
                                                              /Feature_name
     where    d:containsInDistance($item1,$item2)
  </Query>
  <Action>
    <Relation RuleID="R5" Type="satisfiability"
              DocType1="Module Model" DocType2="Feature Model">
      <Element Document="file://c:/ModuleModel.xml">
                                        {$item1/../Module_name} </Element>
      <Element Document="file://c:/Feature_MP.xml">{$item2} </Element>
    </Relation>
  </Action>
</TraceRule>
```

Figure 5- 13: Example of *satisfiability* traceability rule

Rule R6: Implements

As shown in Figure 5-14, this rule can establish an *implements* relation between an operation of a class in a class diagram and a use case when the description of the use case contains the name of the operation and the name of the class of this operation. According to Figure 5-10, examples of *implements* traceability relations from this rule exist (i) between use case *UC3* and operation *takePhoto:void* of class *Camera* and (ii) between use case *UC4* and operation *takePhoto:void* of class *CameraZoom2X*. A relation of type *implements* is created between the names of the class (<Class>) and the name of the operation of the class (<Operation>) in *UML1_PM1.xml* (first <Element>) and the title of use case *UseCase_UC3.xml* (second <Element>). An element <Description> is used to indicate that the relation holds between the operation of the class and the description of the use case.

```
<TraceRule RuleID="R6"  RuleType="implements"
          DocType1="Class Diagram" DocType2="Use Case">
  <Query>
     declare namespace UML="org.omg.xmi.namespace.UML";
     declare namespace d="java:distanceControl.d";

     for $item1 in doc("file://c:/UML1_PM1.xml")//UML:Classifier.feature
                                                    /UML:Operation/@name,
         $item2 in doc("file://c:/UseCase_UC3.xml")//Use_Case
     let $t1 := $item1/../../../@name

     where
        d:containsInDistance($item2/Description, $t1) and
        d:containsInDistance($item2/Description,$item1)

  </Query>
  <Action>
    <Relation RuleID="R6" Type="implements"
              DocType1="Class Diagram" DocType2="Use Case">
      <Element Document= "file://c:/UML1_PM1.xml"> <Class> {$t1} </Class>
                          <Operation> {$item1} </Operation> </Element>
      <Element Document="file://c:/UseCase_UC3.xml"> {$item2/Title} <Description/>
                                                          </Element>
    </Relation>
  </Action>
</TraceRule>
```

Figure 5- 14: Example of *implements* traceability rule

Rule7: Different

This example is one of the rules for identifying a *different* traceability relation which represents the interdependency of a variation point between two product members. As shown in Figure 5-15, this rule can establish a *different* relation between two use cases when there are two *implements* relations between two different use cases and two different classes, and these classes are subclasses of the same superclass. In order to support this case, the rule uses *getParentofVariantClasses()* and *getClassID()* extra functions. As shown in Figure 5-10, an example of a *different* traceability relation from this rule exists between use cases *UC3* and *UC4* that have *implements* relations with the operation *takePhoto:void* of the class *Camera* and the operation *takePhoto:void* of the class *CameraZoom2x*, respectively. Whereas the classes *Camera* and *CameraZoom2x* are subclasses of the class *CameraApplication.*

```
<TraceRule RuleID="R7" RuleType="different"
           DocType1="XML-Based-Rel" DocType2="XML-Based-Rel">
  <Query>
    declare namespace UML="org.omg.xmi.namespace.UML";
    declare function local:getParentClass($child as xs:string) as item()
    {   for $itemA in doc("file://c:/UML1_PM1.xml")//UML:Generalization
                                                    /UML_Generalization.child
        where  $itemA/UML:Class/@xmi.idref = $child
        return  $itemA/../UML:Generalization.parent/UML:Class
    };
    declare function local:getParentofVariantClasses($one as xs:string, $two as
                    xs:string)as item()
    {   for $item1 in doc("file://c:/UML1_PM1.xml")//UML:Generalization
                                                    /UML:Generalization.child,
          $item2 in doc("file://c:/UML1_PM1.xml")//UML:Generalization
                                                    /UML:Generalization.child
        where
          ($item1/UML:Class/@xmi.idref = $one and
           $item2/UML:Class/@xmi.idref = $two and
           local:getParentClass($item1/UML:Class/@xmi.idref ) =
           local:getParentClass($item2/UML:Class/@xmi.idref) and
           local:getParentClass($item1/UML:Class/@xmi.idref) != "" and
           $item1/UML:Class/@xmi.idref != $item2/UML:Class/@xmi.idref )
        return local:getParentClass($item1/UML:Class/@xmi.idref)
    };
    declare function local:getClassID($name as xs:string)as xs:string
    {   for $itemB in doc("file://c:/UML1_PM1.xml")//UML:Class/@name
        where  $itemB = $name
        return  $itemB/../@xmi.id
    };
    for $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="implements"],
        $item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="implements"]
    where
        $item1/@DocType1 ="Class Diagram" and $item1/@DocType2="Use Case" and
        $item2/@DocType1="Class Diagram" and $item2/@DocType2="Use Case" and
       (string($item1/Element[2]/@Document) !=
        string($item2/Element[2]/@Document)) and
       ($item1/Element[1]/@Document = $item2/Element[1]/@Document) and
       ($item1/Element[1]/Class != $item2/Element[1]/Class) and
    local:getParentofVariantClasses(
            local:getClassID(string($item1/Element[1]/Class/@name)),
            local:getClassID(string($item2/Element[1]/Class/@name))) != ""
  </Query>
  <Action>
    <Relation RuleID="R7" Type="different" Term="class implements use case">
      <Element> {$item1/Element[2]/@Document}
                  {$item1/Element[2]/Title}</Element>
      <Element> {$item2/Element[2]/@Document}
                  {$item2/Element[2]/Title}</Element>
      <Implements>{string($item1/Element[1]/Class)}</Implements>
      <Implements>{string($item2/Element[1]/Class)}</Implements>
      <VariantOf>{local:getParentClass(local:getClassID
                          ($item1/Element[1]/Class/@name))}</VariantOf>
    </Relation>  </Action> </TraceRule>
```

Figure 5- 15: Example of *different* traceability rule

Rule R8: Overlaps

This rule can establish an *overlaps* relation between a sequence and class diagram when there is an operation in the sequence diagram with the same name as an operation of a class in a class diagram and the class of the object of the operation in the sequence diagram is the same as the class of the operation in the class diagram. Due to simplication, Figure 5-14 does not show the full declaration of extra functions implemented in XQuery (*getOperationinSeq()*, *getObjectinSeq()*, *getClassObjectinSeq()*, *getClassinClass()*, and *getOperationinClass()*). The explanation and complete declaration of the functions can be found in Section 5.3 and Appendix C, respectively. This rule also uses an XQuery function *string()* to remove whitespace occurred at the beginning and end of the name of an operation. According to Figure 5-10, an example of an *overlaps* traceability relation from this rule exists between operation *takePhoto:void* of class *Camera* and operation *takePhoto:void* in the sequence diagram. The relation is created between the object (<Object>) and the operation (<Operation>) in the sequence diagram represented in *UML1_Pm1.xml* (first <Element>). And the class (<Class>) and the operation (<Operation>) in the class diagram represented in *UML1_Pm1.xml* (second <Element>).

```
<TraceRule RuleID="R8" RuleType="overlaps"
          DocType1="sequence diagram" DocType2="class diagram">
  <Query>
    ...
      let $x1 := local:getOperationinSeq()
        for $x2 in $x1
          for $x3 in local:getObjectinSeq($x2)
            for $x4 in local:getClassObjectinSeq($x3)
              for $x5 in local:getClassinClass($x4, "CD_phone")
                let $x6 := $x5/UML:Classifier.feature/UML:Operation
                  where $x2/../../../../../../@name = "SD_phone"  and
                        (string($x6/@name)) = (string($x2/@name))
  </Query>
  <Action>
    <Relation RuleID="R8" Type="overlaps"
              DocType1="sequence diagram" DocType2="class diagram" >
      <Element Document= "file://c:/UML1_PM1.xml">
                    <Object> $x2/../../../../../../@name </Object>
                      <Operation> $x2/@name </Operation> </Element>
      <Element Document= "file://c:/UML1_PM1.xml">
                    <Class> {$x5/@name} </Class>
                    <Operation> {$x6/@name} </Operation> </Element>
    </Relation>
  </Action>
</TraceRule>
```

Figure 5- 16: Example of *overlaps* traceability rule

Rule 9: Evolution

This rule can establish an *evolution* relation between two state diagrams when the parameters of the same signals are different in the two state diagrams. As shown in Figure 5-17, rule R9 checks (a) if the names of the diagrams being compared are the same, (b) if the names of the transitions are the same, (c) if the names of the actions are the same, (d) if the signal events are the same, (e) if the signals are the same, and (f) if the parameters of the signals are different. An example of an *evolition* traceability relation from this rule is shown in Figure 5-10 between the two extracts of the sequence diagrams. The relation is created between the name of a transition (<Transition>) together with the parameter of the transition (<Parameter>) in *UML1_PM1.xml* (first <Element>) and the name of a transition (<Transition>) together with the parameter of the transition (<Parameter>) in *UML2_PM1*.xml (second <Element>).

```
<TraceRule RuleID="R9"  RuleType="evolution"
            DocType1="Statechart Diagram" DocType2="Statechart Diagram">
  <Query>
      for $item1 in doc("file://c:/UML1_PM1.xml")//UML:Transition,
          $item2 in doc("file://c:/UML2_PM1.xml")//UML:Transition

      where
          $x1/../UML:Diagram/@name = $x2/../ UML:Diagram/@name  and
          $x1/@name = $x2/@name and
          $x1/UML:Transition.effect/UML:ActionSequence/UML:ActionSequence.action
                                                  /UML:UninterpretedAction/@name =
          $x2/UML:Transition.effect/UML:ActionSequence/UML:ActionSequence.action
                                                  /UML:UninterpretedAction/@name and
          $x1/UML:Transition.trigger/Behavioral_Elements.State_Machines.Event
                      /@xmi.idref =  $x1/../UML:SignalEvent/@xmi.id and
          $x2/UML:Transition.trigger/Behavioral_Elements.State_Machines.Event
                      /@xmi.idref =  $x2/../UML:SignalEvent/@xmi.id and
          $x1/../UML:SignalEvent/@name = $x2/../UML:SignalEvent/@name and
          $x1/../UML:SignalEvent/UML:SignalEvent.signal/Behavioral_Elements.Common
          _Behavior.Signal/@xmi.idref = $x1/../UML:Signal/@xmi.id and
          $x2/../UML:SignalEvent/UML:SignalEvent.signal/Behavioral_Elements.Common
          _Behavior.Signal/@xmi.idref = $x2/../UML:Signal/@xmi.id and
          $x1/../UML:Signal/@name = $x2/../UML:Signal/@name and
          $x1/../UML:Signal/@xmi.id/UML:DataType/@xmi.id =
          $x1/../UML:Event.Parameter/UML:Parameter/UML:Parameter.type
                      /Foundation.Core.Classifier/@xmi.idref and
          $x2/../UML:Signal/@xmi.id/UML:DataType/@xmi.id =
          $x2/../UML:Event.Parameter/UML:Parameter/UML:Parameter.type
                      /Foundation.Core.Classifier/@xmi.idrefand
          $x1/../UML:Event.Parameter/UML:Parameter/@name !=
          $x2/../UML:Event.Parameter/UML:Parameter/@name

  </Query>
  <Action>
    <Relation RuleID="R9" Type="evolution"
              DocType1="Statechart Diagram" DocType2="Statechart Diagram">
      <Element Document="file://c:/UML1_PM1.xml">
        <Transition>{$x1/@name} </Transition>
        <Parameter> {$x1/../UML:Event.Parameter/UML:Parameter/@name }
                      </Parameter>
      </Element>
      <Element Document="file://c:/UML2_PM1.xml">
        <Transition> {$x2/@name } </Transition>
        <Parameter> {$x2/../UML:Event.Parameter/UML:Parameter/@name}
                      </Parameter>
      </Element>
    </Relation>
  </Action>
</TraceRule>
```

Figure 5- 17: Example of *overlaps* traceability rule

Additionally, we have developed 63 traceability rule templates from which 51 are for direct traceability relations and 12 are for indirect traceability relations (see in Appendix B). We have used all those 63 traceability rule templates in our experiments (see in Chapter 8). The following subsection describes the extended functions that we have implemented to support the traceability rules.

5.3. Extended Functions

The satisfaction of all possible conditions for the traceability relations such as considering grammatical structures of the sentences in the documents and synonyms requires a need for extra functions to allow the traceability generator to identify the relations. The extended functions can be classified in two main groups. One group is concerned with functions that have been implemented in XQuery (viz. XQuery and the other group with functions implemented in java (viz. java functions). Additionally, XQuery language allows us to add new functions and commands in its own and other languages. Extra functions implemented in XQuery are concerned with the retrieval of specific elements in the documents, whereas the functions implemented in java are concerned with the manipulation with textual aspects in the documents. We present below the extended functions that are implemented in two languages i.e. in XQuery and in Java.

5.3.1. Functions in XQuery

As mentioned, the extended functions represented in XQuery are concerned with functions that identify specific elements in the documents used in our approach such as state in a statechart diagram, classes in a class diagram, messages and objects in a sequence diagram, and features in a feature model. Generally, an XQuery function declaration is composed of two parts as shown in Figure 5-18.

Part I: It is concerned with the name and signature of the function. It contains:

(i) *declare function* keywords to start the declaration of the function;

(ii) *local:* keyword which defines the scope of a function;

(iii) the name of the function; and

(iv) the input and output parameters of the function when applicable.

Part II: It is concerned with the definition of the function (*function body*), which consists of XQuery statements. The function body can include built-in XQuery functions or other extended functions implemented in XQuery or Java.

```
declare function local:function-name($variable-name as input)
                                                    as output
{
      function-body
};
```

Figure 5- 18: A structure of a user-defined function

In the following, we present each of the extra XQuery functions. A complete declaration of these functions with their respective body is shown in Appendix C.

I. getTransitioninState

```
declare function local:getTransitioninState() as item()*
```

Figure 5- 19: getTransitioninState function

The *getTransitioninState* function identifies the set of transitions in a statechart diagram (Figure 5-19). The function does not take any input parameters and returns a sequence of one or more transitions (that appear in XMI[5] document as UML:Transition elements) in a statechart diagram. The result of this function appears as item() in XQuery. For instance, according to the extract of a statechart diagram (shown in Figure 5-20), the getTransitioninState function returns a set of transitions – {transition *a*, transition *b*, transition *c*}.

[5] See details in XMI specification OMG. XMI.

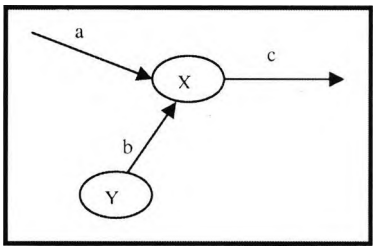


Figure 5- 20: Extract of a statechart diagram

II. getStateinState

```
declare function local:getStateinState($transition as node())
                                        as item()
```

Figure 5- 21: getStateinState function

The *getStateinState* function identifies the state of a transition in a statechart diagram (Figure 5-21). The function takes a transition as an input (appearing in an XMI document as an UML:Transition element) and returns the state (appearing as an UML:SimpleState element) of the transition. The result of this function appears as item() in XQuery. According to the previous example (Figure 5-20), the *getStateinState* function for transitions *a* or *b* returns state *X*.

III. getMessageinSeq

```
declare function local:getMessageinSeq() as item()*
```

Figure 5- 22: getTransitioninState function

The *getMessageinSeq* function identifies the set of messages in a sequence diagram (Figure 5-22). The function takes no input parameter and returns a sequence of one or more messages (appearing in an XMI document as an UML:Link element) in a sequence diagram. The result of this function appears as item()* in XQuery. According to an example of a sequence diagram in Figure 5-23, the getMessageinSeq function returns a set of messages – {message *a*, message *b*, message *c*}.

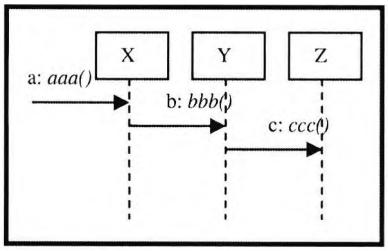


Figure 5- 23: Extract of a sequence diagram

IV. getObjectinSeq

```
declare function local:getObjectinSeq($link as node()) as
                                        item()
```

Figure 5- 24: getObjectinSeq function

The *getObjectinSeq* function identifies the object that receives a message in a sequence diagram (Figure 5-24). In general a message in a sequence diagram represents the communication between two objects. The function takes a message as an input parameter (appearing in an XMI document as an UML:Link element) and returns the object (appearing in the XMI document as an UML:Object element) that receives the message. The result of this function appears as item() in XQuery. According to the example in Figure 5-23, the getObjectinSeq function returns object *Y* (as the UML:Object element) for message *b* as input parameter.

V. getClassObjectinSeq

```
declare function local:getClassObjectinSeq($object as node())
                                        as item()
```

Figure 5- 25: getClassObjectinSeq function

The *getClassObjectinSeq* function identifies the class of an object in a sequence diagram (Figure 5-25). The function takes an object as the input parameter and returns its class (as appearing as item() in XQuery). According to the example in Figure 5-23, the getClassObjectinSeq function takes object *Y* as input parameter and returns class *Y'* (as the UML:Class element) whereby the object is instantiated.

VI. getClassinClass

```
declare function local:getClassinClass($diagram as xs:string)
                                        as item()*
```

Figure 5- 26: getClassinClass function

The *getClassinClass* function identifies classes in a class diagram (Figure 5-26). The function takes as input parameter the name of the class diagram, and returns the classes in the diagram (appearing as UML:Class elements). The result of this function appears as item()* in XQuery. For instance, according to the extract of a class diagram (shown in Figure 5-27), the *getClassinClass* function returns a set of classes – {class *C0*, class *C1*, class *C2*}.

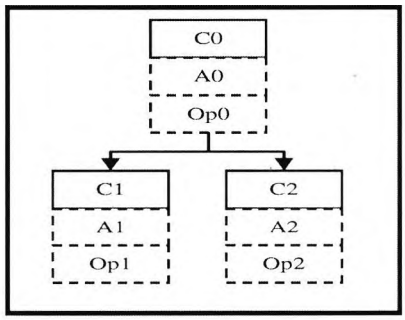


Figure 5- 27: Extract of a class diagram

VII. getParentFeature

```
declare function local:getParentFeature($child as xs:string) as
                                        item()
```

Figure 5- 28: getParentFeature function

The *getParentFeature* function identifies the parent feature of a feature in a feature model (Figure 5-28). The function takes as input a feature name and returns the name of its parent feature, if available. The output of the function is a sequence of words with part-of-speech tags as item() in XQuery. For example, Figure 5-29 shows an extract of a feature model. In the figure, feature *n* has a parent feature *m* and children features *h* and *i*. Moreover, feature *m* has another child feature *l* which

has children features *j* and *k*. When the function has feature *n* as its input parameter, it returns feature name *m* as the parent feature of feature *n*.

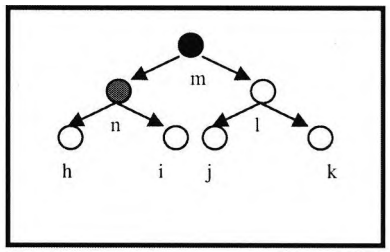


Figure 5- 29: Extract of feature model

VIII. getChildrenFeature

```
declare function local:getChildrenFeature($parent as xs:string)
                                          as item()*
```

Figure 5- 30: getChildrenFeature function

The *getChildrenFeature* function identifies the set of children features of a feature in a feature model (Figure 5-30). The function takes a feature name as its input parameter and returns the sequence of children-feature names as item()* in XQuery, if available. The output of the feature is a sequence of words with part-of-speech tags. According to the previous example, when function *getChildrenFeature* has feature name *n* as its input parameter, it returns the sequence of children-feature *h* and *i*.

IX. getFeatureofSubsystem

```
declare function local:getFeatureofSubsystem($subsystem as
                                    xs:string) as item()*
```

Figure 5- 31: getFeatureofSubsystem function

The *getFeatureofSubsystem* function identifies the set of features used by a subsystem in a subsystem model (Figure 5-31). The function takes as input parameter the name of a subsystem and returns a sequence of one or more feature names as item()* in

XQuery, if available. An example of the use of this function is illustrated by considering Figure 5-9. When function *getFeatureofSubsystem* has subsystem name *Messaging*, it returns feature *Text Messages*.

X. getOperationinSeq

```
declare function local:getOperationinSeq() as item()*
```

Figure 5- 32: getOperationinSeq function

The *getOperationinSeq* function identifies the set of operations in a sequence diagram (Figure 5-32). The function does not take any input parameter and returns the operations (appearing in XMI document as UML:Operation elements) in the diagram. The result of this function appears as item()* in XQuery. An example of the use of this function is illustrated by considering Figure 5-23. The function *getOperationinSeq* returns operations *aaa()*, *bbb()*, and *ccc()*.

XI. getOperationinClass

```
declare function local:getOperationinClass ($class as node())
                                            as item()*
```

Figure 5- 33: getOperationinClass function

The *getOperationinClass* function identifies the set of operations of a class in a class diagram (Figure 5-33). The function takes a class as its input parameter and returns a set of operations of the class as item()* in XQuery. An example of the use of this function is illustrated by considering Figure 5-27. The function *getOperationinClass* has class *C1* and returns operation *Op1()*.

XII. getStateofOperationinState

```
declare function local:getStateofOperationinState($operation as
                                        node()) as item()
```

Figure 5- 34: getStateofOperationinState function

The *getStateofOperationinState* function identifies the state in a statechart diagram that receives an event when the event represents an operation (Figure 5-34). The function takes an operation (appearing in XMI document as an UML:Operation element) and returns a sequence of one or more states (appearing as UML:SimpleState elements) in statechart diagram of which name are given as the particular operation name. The result of this function appears as item() in XQuery.

XIII. getParentofVariantFeatures

```
declare function local:getParentofVariantFeatures($one as
                    node(), $two as node()) as item()
```

Figure 5- 35: getParentofVariantFeatures function

The *getParentofVariantFeatures* function identifies the parent feature of two features that are either alternative or optional feature in a feature model (Figure 5-35). The function takes as input parameters the names of two features and returns a node representing the parent feature in a feature model as item() in XQuery, when these features are alternative or optional features and they have the same parent feature. According to the example in Figure 5-29, when function *getParentofVariantFeature* has alternative features *j* and *k* as its input parameters, it returns the parent feature *l*.

XIV. getParentofVariantClasses

```
declare function local:getParentofVariantClasses($one as
                    xs:string, $two as xs:string) as
                    item()
```

Figure 5- 36: getParentofVariantClasses function

The *getParentofVariantClasses* function identifies the generalized class (superclass) of two classes that are specialized (subclass) (Figure 5-36). The function takes as input parameters the names of two classes and returns a node representing the generalized class in a class diagram as item() in XQuery, if available. As shown in Figure 5-27,

the function getParentofVariantClasses has the names of classes *C1* and *C2* and returns class *C0*.

XV. getParentClass

```
declare function local:getParentClass($child as xs:string) as
                                        item()
```

Figure 5- 37: getParentClass function

The *getParentClass* function identifies the generalized class of a class in a class diagram (Figure 5-37). The function takes as input parameter the name of a class and returns a node representing the generalized class in a class diagram as item() in XQuery, if available. According to the example in Figure 5-27, the function getParentClass has the name of class *C1* and returns class *C0*.

XVI. getClassID

```
declare function local:getClassID($name as xs:string) as
                                    xs:string
```

Figure 5- 38: getClassID function

The *getClassID* function identifies the identifier of a class in a class diagram (Figure 5-38). The function takes as input parameter the name if a class and returns a string representing the identifier (ID) of the class.

5.3.2. Functions In Java

As mentioned, the extended functions implemented in java are concerned with functions that manipulate with textual aspects in the documents. More specifically, these functions tackle specific string manipulation issues such as synonyms, distance between words in a sentence or paragraph, and concatenation of string.

Generally, a java function is specified as part of a package. In our work, we have implemented two packages, namely package *synonym* that contains function

findSynonym, and package *distanceControl* that contains functions *containsInDistance*, *stringNoSpace*, *setoff*, and *checkDistanceControl*. The declaration of a java function is shown in Figure 5.39. As shown in the figure, the java function is declared as *namespace* and attributed to a variable namespace.

```
declare namespace variable-namespace ="java:package.class";
```

Figure 5- 39: the declaration of a namespace referring to extra functions in Java package

Calling a function can be appeared in the second and third subparts of Part 2 of the traceability rules (as shown in Figure 5-3). It contains the variable-namespace which is used as a prefix for extra functions, followed by a colon ":", a name of function, and a parameter(s) (if required). The template of calling an extended function implemented in java is shown in below figure.

```
variable-namespace: function-name($variable-name as input)
                                as output
```

Figure 5- 40: Calling an extended function implemented in Java

Currently we have developed five java functions as presented below. In the following, the description of the java functions present how these functions are called in XQuery statements and the signature of these functions in java. The complete codes of these functions are described in Appendix Y.

I. containsInDistance()

The *containsInDistance* function is a Boolean function that determines if a text contains certain words, or their synonyms, at an appropriate distance and considers their part-of-speech. The text may be paragraph, a sentence, or words annotated with POS-XML tags. An appropriate distance means the existence of words appearing in the same sentence. As shown in Table 5-1, there are variation of this

function with different types and numbers (two or three) of parameters. In java, the parameters can be of type *string* (a word or many words in a text that is recognized in XQuery language as xs:string), *arraylist* [6](a textual paragraph with POS-XML tags which appears as a sequence of XML elements in a XML-formatted document and recognized in XQuery language as xs:element[7]), and *object*[8] (a textual paragraph with POS-XML tags which appears as a sequence of XML nodes in a XML-formatted document and is recognized in Xquery language as xs:node[9]).

Table 5- 1: Variation of *containsInDistance* function with different parameters

	Xquery statement	Java signature
a)	*s*: **containsInDistance**($*words1* as node(), $*words2* as xs:string, $*words3* as xs:string) as as:Boolean	public static *boolean* **containsInDistance** (*Object* words1, *String* words2, *String* words3)
b)	*s*: **containsInDistance**($*words1* as node(), $*words2* as xs:string) as xs:boolean	public static *boolean* **containsInDistance** (*Object* words1, *String* words2)
c)	*s*: **containsInDistance**($*words1* as element()*, $*words2* as xs:string) as xs:boolean	public static *boolean* **containsInDistance** (*ArrayList* words1, *String* words2)
d)	**containsInDistance**($*words1* as node(), $*words2* as node(), $words3 as element()*) as xs:Boolean	public static *boolean* **containsOInDistance** (*Object* words1, *Object* words2, *ArrayList* words3)
e)	*s*: **containsInDistance**($*words1* as node(), $*words2* as node()) as xs:boolean	public static *boolean* **containsOInDistance** (*Object* words1, *Object* words2)

[6] Java *arrayList* is applied to represent a sequence of an XML node or an XML element
[7] see details in W3C. XQuery.
[8] Java *Object* is applied to represent an XML node or an XML element
[9] see details in W3C. XQuery.

II. findSynonym()

The *findSynonym* function identifies a set of synonyms for a word. As shown in Table 5-2, this function taka word as an input parameter of type string in Java and returns a set of synonyms represented as arrayList in Java, or null if the word has no synonyms. The synonyms are identified based on WordNet (WordNet).

Table 5- 2: A layout of *findSynonym* function

XQuery statement	Java signature
s: **findSynonym**($*word* as xs:string)as element()*	public static *ArrayList* **findSynonym** (*String* strInput)

III. stringnospace()

The *stringnospace* function returns a string without white spaces. As shown in Table 5-3, the function takes a sequence of words as its input parameter and returns a string with the concatenation of these words without whitespaces.

Table 5- 3: A layout of *stringnospace* function

XQuery statement	Java signature
s: **stringnospace**($*words* as xs:string)as xs:string	public static *String* **stringnospace**(*String* str)

IV. setof()

The *setof* function returns a set of words composed by the input parameters. As shown in Table 5-4, the function takes four parameters of array lists and returns a composed array list.

Table 5- 4: A layout of *setof* function

XQuery statement	Java signature
s: **setof**($*s1* as element()*, $*s2* as element()*, $*s3* as element()*, $*s4* as element()*) as item()*	public static *ArrayList* **setof**(*ArrayList* s1, *ArrayList* s2, *ArrayList* s3, *ArrayList* s4)

V. checkDistanceControl()

The *checkDistanceControl* function is a Boolean function that identifies if the set of synonyms of two words are associated in a textual paragraph. More specifically, the paragraph contains two words, or ones of their synonyms and the existence of these words must appear in the same sentence in the paragraph. The function takes as input parameters a text and two sets of synonyms and returns true if the text contains some of the words in the two sets of synonyms at an appropriate distance. As shown in Table 5-5, the function takes three input parameters: first is typed of string in Java and also recognized as xs:string in XQuery language; second and third are arrayList in Java and xml elements in XQuery.

Table 5- 5: A layout of *checkDistanceControl* function

XQuery statement	Java signature
s: **checkDistanceControl** ($*longstring* as xs:string, s1 as element()*, s2 as element()*) as xs:boolean	public static *boolean* **checkDistanceControl**(*String* Description, *ArrayList* s1, *ArrayList* s2)

5.4. Summary

This chapter has presented the traceability framework to support traceability generation in product family systems. It also described and gave examples of different types of traceability rules. In addition, it presented the various types of extra functions that we have implemented.

Chapter 6

XTraQue Tool

This chapter presents the prototype tool called *XTraQue* that we have implemented to demonstrate and evaluate our work. It aims to illustrate how the XTraQue tool can facilitate the traceability activity by generating traceability relations according to the traceability reference model in Chapter 4. Section 6.1 describes the overview and functionalities of the XTraQue tool. Section 6.2 presents the interfaces of the tool. Section 6.3 summarises the chapter.

6.1. Overview

In order to evaluate and demonstrate our approach, we have implemented a prototype tool called XTraQue. We envisage the use of our tool as a general platform for automatic generation of traceability relations and support for product family system development. The tool has been implemented in Java and uses Saxon to evaluate XQuery statements. The extra functions not supported by XQuery, as described in Section 5.3., have also been implemented in Java.

Figure 6-1 illustrates the architecture of XTraQue tool. The tool is composed of seven components, namely:

(a) *XTraQue Interface* – This component provides the user interfaces for a user to interact with the tool ranging from the types of documents to be traced, to the types of relations, and family of products.

(b) *Document Identifier* – This component identifies a set of relevant documents to be traced based on the input from the *XTraQue Interface* component.

(c) *Rule Template Identifier* – This component identifies a set of traceability rule templates that are related to the documents and relations to be traced based on inputs to the *XTraQue Interface* component.

(d) *Traceability Rule Editor* – This component verifies XQuery statements in traceability rule templates and displays the results of the XQuery statements.

(e) *Rule Instantiator* – This component creates a set of instantiated traceability rules by replacing the placeholders of the document types in the identified traceability rule templates with the names of the documents to be traced. The identified traceability rule templates and the names of the documents are derived from the *Rule Template Identifier* and *Document Identifier*, respectively.

(f) *Traceability Relation Generator* – This component generates traceability relations by executing the traceability rules created by the *Rule Instantiator.*

(g) *Traceability Relation Presenter* – This component records and presents the traceability relations generated by the *Traceability Relation Generator.*

According to Figure 6-1, the *XTraQue Interface* component is responsible for communication with a user for: (a) specifying the criteria for traceability generation; and (b) entering a new traceability rule to be verified. Consider in Case (a), the *Document Identifier* component identifies a set of documents and the *Rule Template Identifier* identifies a set of traceability rule templates corresponding the criteria derived from the *XTraQue Interface* component. The *Rule Instantiator* component is responsible for creating a set of instantiated rules by instantiating the placeholders for the documents identified by the *Document Identifier* component in the rule templates identified by the *Rule Template Identifier* component. The *Traceability Relation Generator* component is responsible for (i) executing the instantiated rules created by the *Rule Instantiator* component and extra functions used in the rules; and (ii) generating traceability relations between the identified documents. The *Traceability Relation Presenter* component is responsible for recording and representing the relations created by the *Traceability Relation Generator* component in XML documents. However, consider Case (b), the *Traceability Rule Editor* component is responsible for

(i) verifying XQuery statements in traceability rule templates derived from the *XTraQue Interface* component and (ii) displaying its results.

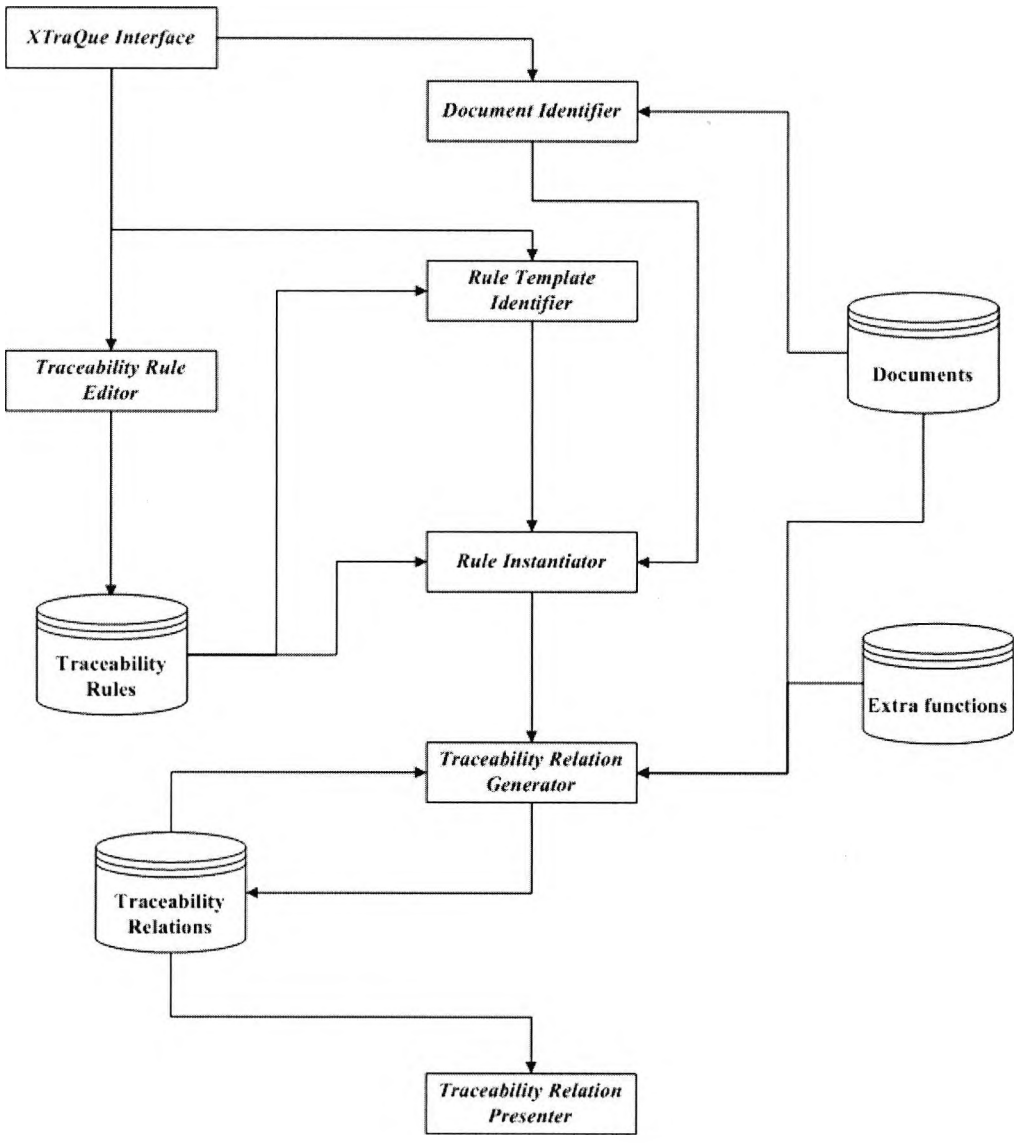


Figure 6- 1: The Architecture of XTraQue Tool

The various components of the XTraQue tools support various functionalities. These functionalities include:

(i) *Traceability Selection*, which is concerned with the specification of the documents to be traced and the types of relations to be created;

(ii) *Traceability Creation*, which is concerned with the generation of direct and indirect traceability relations based on the input given in (i);

(iii) *Traceability Presenter,* which is concerned with the recording and representation of the traceability relations generated in (ii); and

(iv) *Trace Rule Editor,* which is concerned with the visualization and testing of new traceability rules.

In the following, we explain these functionalities in more details.

I. Traceability Selection

In order to support this functionality, the tool provides sophisticated user interfaces in which a user can select to establish traceability relations between different levels of documents. According to the traceability reference model (in Chapter 4), two levels of documents are namely *product line level* and *product member level.* More specifically, this functionality allows the user to create the relations between:

(a) documents of two specific product members,

(b) documents at the level of product line and one specific product member, and

(c) documents at the level of product line and two specific product members.

Case (a) supports the generation of traceability relations in groups 2, 3, 4, and 5, as described in Chapter 4. Case (b) supports the generation of traceability relations in groups 1, 3, 5, and 6. Case (c) supports the generation of traceability relations in all groups.

For any of cases (a) to (c) above, the user can select to trace all the documents related to the levels of product line and product members, or to specify which documents to be traced based on:

- type of documents (e.g. all use cases, class, statechart, and sequence diagrams of the product member level, or all feature, subsystem, process, and module models of the product line level);

- particular document names; or
- types of traceability relations.

In this latest case, the types of documents to be traced are selected depending on the documents that can be associated with a specific relation type. For example, an *implements* relation may exist between class diagram and feature models or use cases, sequence diagrams and feature models or use cases, and statechart diagram and feature models or use cases. Therefore, documents created during domain design at the product line level (i.e. subsystem, process, and module models) will not be selected to be traced in this case. Moreover, the tool will not attempt to establish *implement* relations between documents that have been selected but do not hold the relation type (e.g. feature model and use case).

In the case that the user selects to trace all documents or documents based on first two cases above (type of documents and particular document names), the tool also allows the user to specify the types of relations to be traced. The user can select to trace the documents for all traceability relations for any of those two cases. In Section 6.2, we show user interfaces for this functionality and an example of using the interfaces in which the traceability user has selected the types of artefacts to be traced and the types of traceability relations.

II. Traceability Creation

The generation of direct and indirect traceability relations is executed by the components in XTraQue tool i.e. *Document Identifier, Rule Template Identifier, Rule Instantiator,* and *Traceability Relation Generator.* This functionality reflects *Traceability Generator* process described in Section 5.1. For each pair of documents, the tool identifies traceability rule templates associated with the documents, instantiates the placeholders for the document types in the rule templates, generates direct relations in XML format and indirect relations based on the direct ones also in XML format.

The tool also applies the extra functions implemented in XQuery and Java languages, if applicable.

Since the XTraQue tool allows a user to select to trace all documents of product family systems or specific documents, the tool can generate traceability relations in different levels of granularity, namely:

- at the level of two product members;
- at the level of product line and product member(s);
- at the level of particular documents in the product line; and
- at the level of particular documents in product member(s).

III. Traceability Presenter

The generated traceability relations are recorded and represented in XML documents. Moreover, the XTraQue tool also keeps track of the traceability activity by specific log files represented as XML documents.

IV. Traceability Rule Editor

The XTraQue tool allows for the creation of new traceability rules and the execution of these rules in order to verify their correctness before including these rules in the set of traceability rule templates to be used by the tool. After the traceability user is satisfied with a new rule, this rule can be inserted in the document containing all the traceability rule templates. In Section 6.2, we show a user interface for this functionality and an example of using the interface for the case in which the user has created a new rule and verified its correctness.

6.2. User Interfaces

This section illustrates the user interfaces of the XTraQue tool and describes how a user can execute the various activities supported by the tool. We illustrate the use of the tool by giving examples based on the mobile-phone systems (see Chapter 7).

6.2.1. Specifying the Scope of Traceability Generation

As shown in Figure 6-2, this interface supports the functionality of *Traceability Selection*. The interface consists of three main parts:

(a) a panel representing a product family with its various product members (in this case, a mobile-phone family with three product members i.e. PM1, PM2, and PM3);

(b) a panel consisting of three menus of options for specifying the scope of traceability activity; and

(c) a command button ("Go") for moving to the next interface.

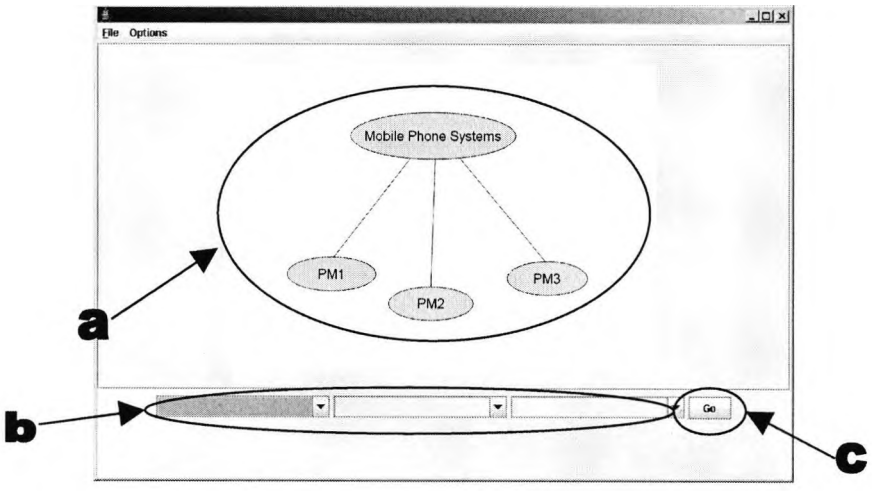


Figure 6- 2: An XTraQue interface for specifying a scope of traceability generation

The panel *b* is for specifying the scope of traceability generation. According to the traceability reference model, the product family systems consist of two levels: product line and product member.

- The first menu of options in the panel is for specifying traceability *between product members* or *between product line and product member(s)* (see Figure 6-3).
- The other two menus of options are for specifying the names of product members to be traced (see Figure 6-3).

The various options in panel *b* allows a traceability user to specify the scope of traceability (i) between one product line and one product member, (ii) between one product line and two product members, or (iii) between two product members.

As shown in Figure 6-3, the user firstly considers the situation in which wants to generate traceability relations between documents at the levels of product line and two product members. Secondly, the user specifies two product members, namely *model PM1* and *model PM2*. Then the user selects to move to the next interface.

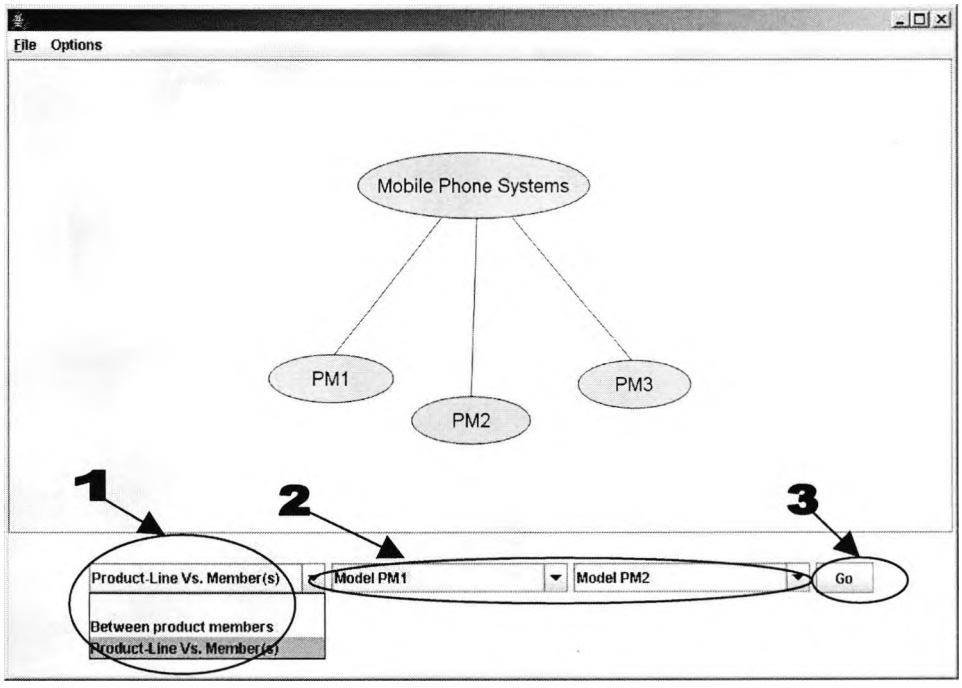


Figure 6- 3: Example interface demonstrating specifying the scope of traceability generation between documents at the levels of product line and two product members, *model PM1* and *model PM2*

6.2.2. Specifying Types of Documents and Relationships

As shown in Figure 6-4, this interface also supports the functionality of *Traceability Selection*. The interface follows from the interface in Figure 6-2. It allows a user to specify the types of documents and relationships to be traced and consists of four main parts:

(a) a panel that is composed of three sub-panels, namely *requirement*, *design*, and *architecture*. Each sub-panel contains different icons representing the various types of documents of each development phase. The *requirement* sub-panel contains *use case* and *feature model* icons representing documents produced during the analysis phase. The *design* sub-panel contains *class diagram*, *statechart diagram* and *sequence diagram* icons representing documents produced during the design phase of the product-members. The *architecture* sub-panel contains *subsystem model*, *process model* and *module model* icons representing documents produced during the design phase of the product-line. Table 6-1 shows all the icons representing the various documents.

(b) a list that shows nine types of traceability relations supported by the approach. This allows the traceability user to select one or many relations types to be generated.

(c) a panel that shows the selected documents to be traced.

(d) a panel with two buttons "Go" and "Trace All", which either presents the next interface or executes the traceability generation for the selected documents and relations, respectively.

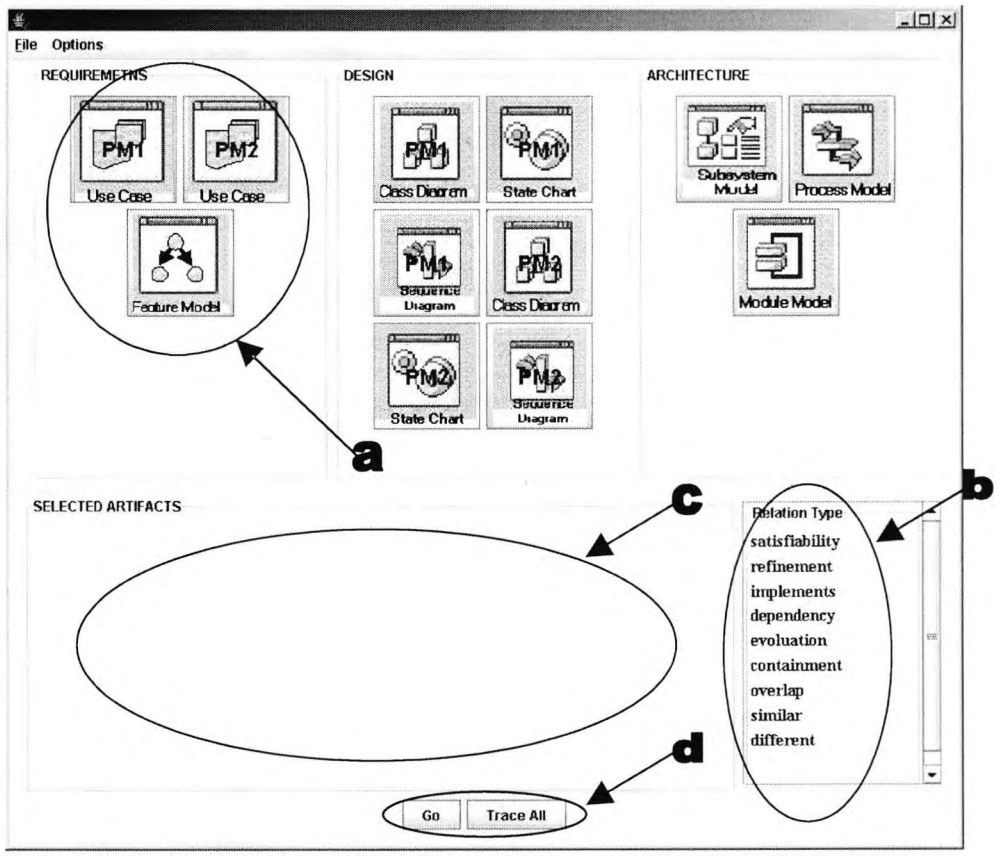
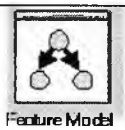
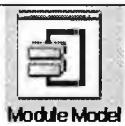
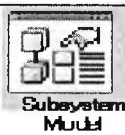


Figure 6- 4: An XTraQue interface for specifying types of document artifacts and relationships according to tracing between the product-line and two product members

The information shown in panel *a* depends on the scope of traceability generation that the user has specified in the previous interface (Figure 6-2). According to the example in Figure 6-3, the scope of traceability generation is specified between a product line and two product members, *model PM1* and *model PM2*. Thus, in this case, panel *a* (Figure 6-4) shows documents in all three sub-panels i.e. *requirements, design,* and *architecture.*

As shown in Figure 6-4, there are three icons (*use case PM1*, *use case PM2* and *feature model*) in sub-panel *requirements*, six icons (*class diagram PM1, class diagram PM2, statechart diagram PM1, statechart diagram PM2, sequence diagram PM1*, and *sequence diagram PM2)* in sub-panel *design*, and three icons (*subsystem model, process model,* and *module model)* in sub-panel *architecture.*

Table 6- 1: Icons in panel (a)

Sub-panel	Icon	Documents
Requirements		Use case
Requirements		Feature model
Design		Class diagram
Design		Sequence diagram
Design		Statechart diagram
Architecture		Module model
Architecture		Process model
Architecture		Subsystem model

Consider the situation in which the scope of traceability generation is between two product members, *model PM1* and *model PM2*. In this case, panel *a* shows only two sub-panels *requirements* and *design* (as shown in Figure 6-5), with icons *use case PM1*, *use case PM2*, *class diagram PM1*, *class diagram PM2*, *statechart diagram PM1*, *statechart diagram PM2*, *sequence diagram PM1*, and *sequence diagram PM2*.

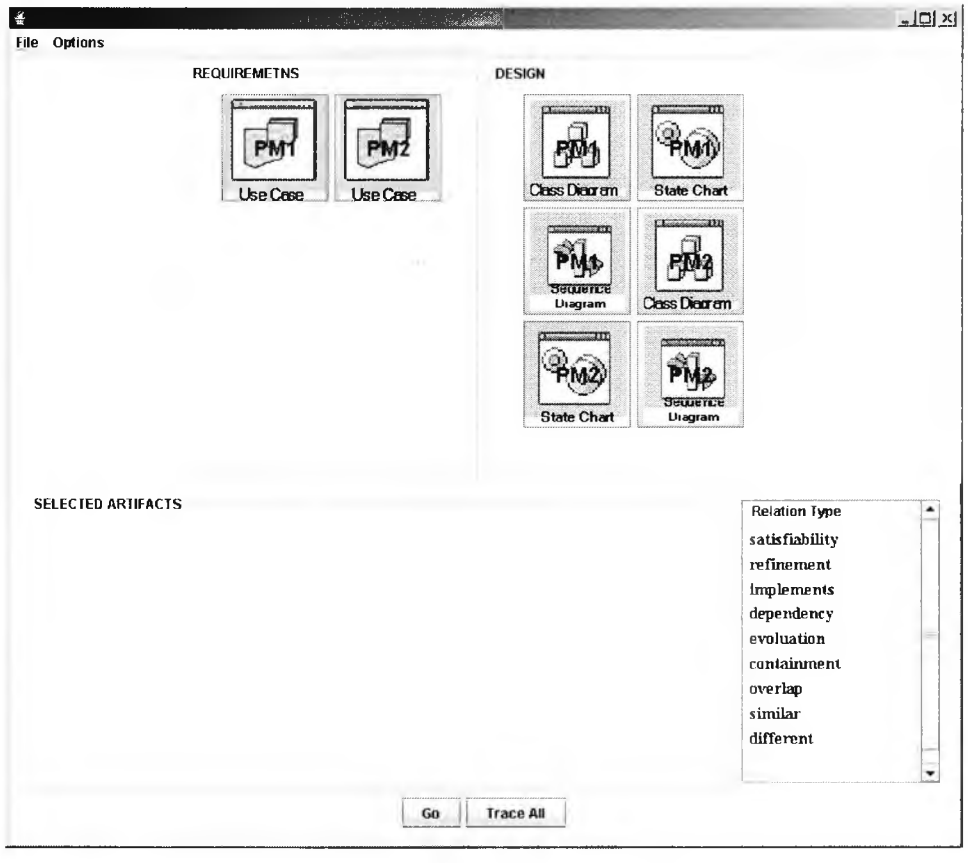


Figure 6- 5: An XTraQue interface for specifying types of document artifacts and relationships according to tracing between two product members, *model PM1* and *model PM2*

Figure 6-6 illustrates how to work with the interface.

- Firstly, the user selects a type of document by clicking on its respective icon in panel *a*, which is then displayed in panel *c*.
- Secondly, the user selects one or many relationship type(s) from the list in panel *b*.
- Next, the user either selects the "Go" button to move to next interface or the "Trace All" button to execute the generation of traceability relations for all selected documents according to the specified criteria (i.e. systems, document types, and relationship types)

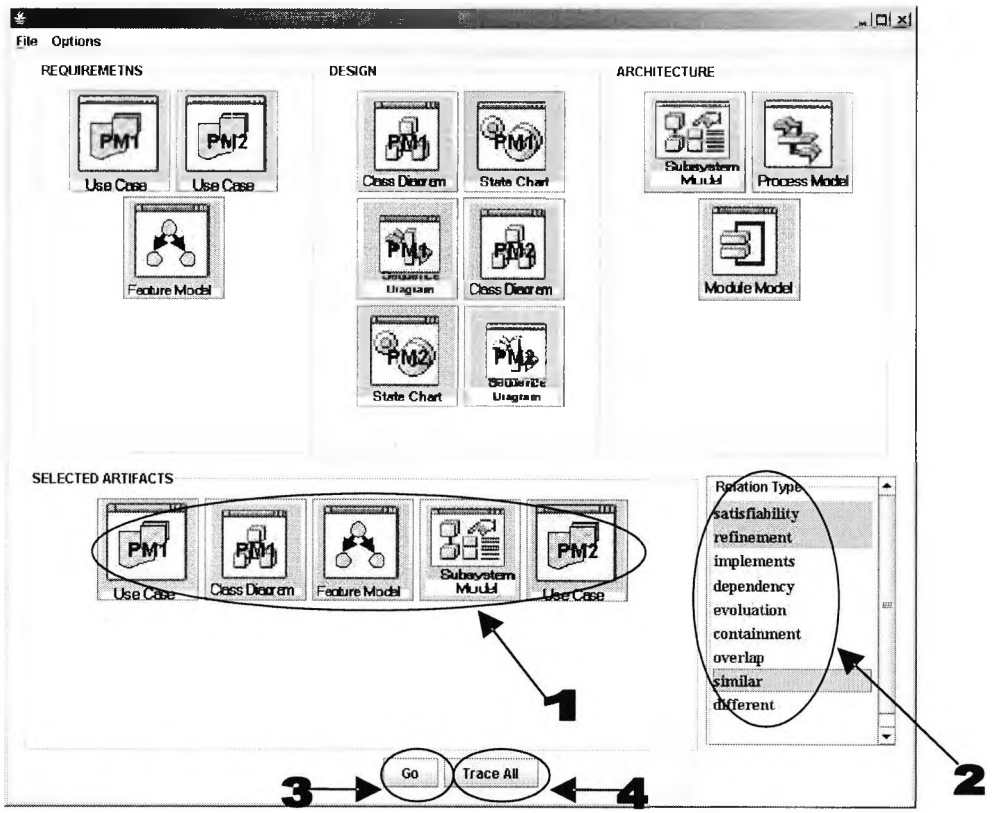


Figure 6- 6: Example interface demonstrating specifying of types of document artifacts and relationships

6.2.3. Specifying Particular Documents and Relationship Types

As shown in Figure 6-7, this interface also supports the functionality *Traceability Selection*. It follows from the interface in Figure 6-4 and allows a user to specify and visualize specific documents of the types selected in the previous interface. This interface consists of:

(a) a panel that lists the types of documents following from the selected documents from the previous interface (Figure 6-6). The lists of documents are categorized as *product line* and *product member* levels. The example in Figure 6-7 shows three lists of documents, namely *product line*, *PM1*, and *PM2*. The *product line* list has two types of documents, feature model and subsystem model. The *PM1* list has two types of documents, use case and class diagram. The *PM2* list has one type of documents, use case; since these have been the documents selected in the previous interface (see Figure 6-6).

(b) a panel that displays lists of document names of the document types shown in panel *a* to be selected by the user. The selected documents from panel b are then listed in a panel *e*.

(c) a panel that displays the content of a selected document.

(d) a list that shows nine types of relationships. This, again, allows a user to specify the types of relationships to be generated.

(e) a panel that shows selected documents to be traced.

(f) a panel with three buttons, namely "Trace", "Reset", and "Save Trace", which are related to actions for generating traceability relations, resetting the selection, and saving the selection, respectively.

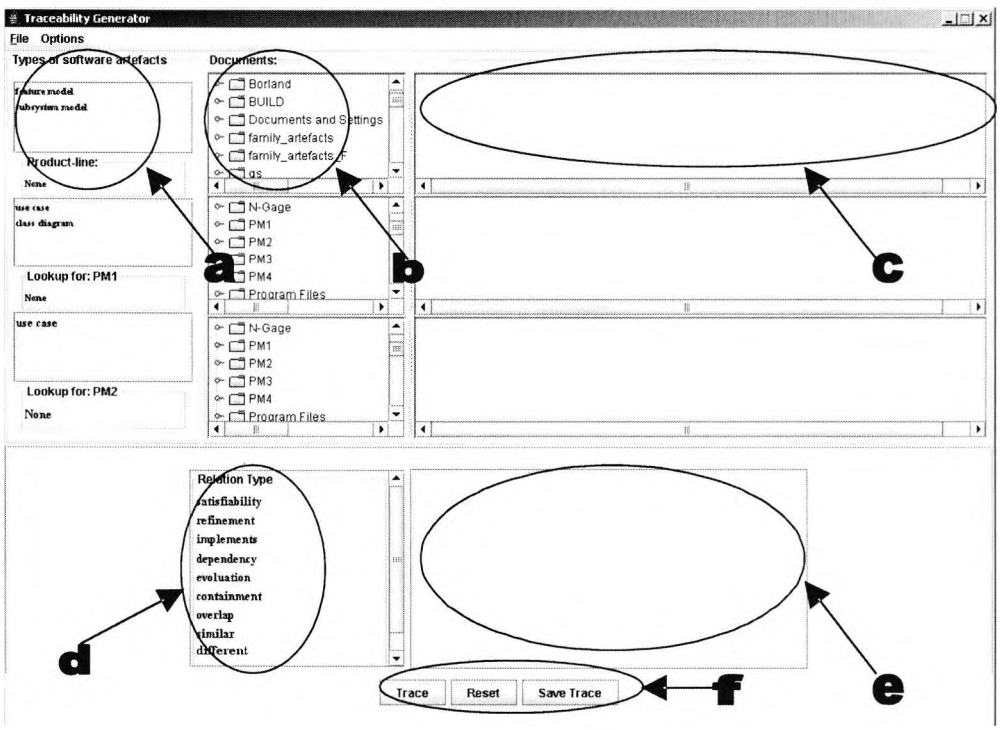


Figure 6- 7: An XTraQue interface for specifying particular documents and relationships according to the specified criteria from the previous interface (Figure 6-4)

The example shown in Figure 6-7 follows from the documents selected in the example in Figure 6-6. This shows the case in which the user has selected documents *use case* and *class diagram* for product member *PM1*, *use case* for product member *PM2*, *feature model* and *subsystem model* from the previous interface.

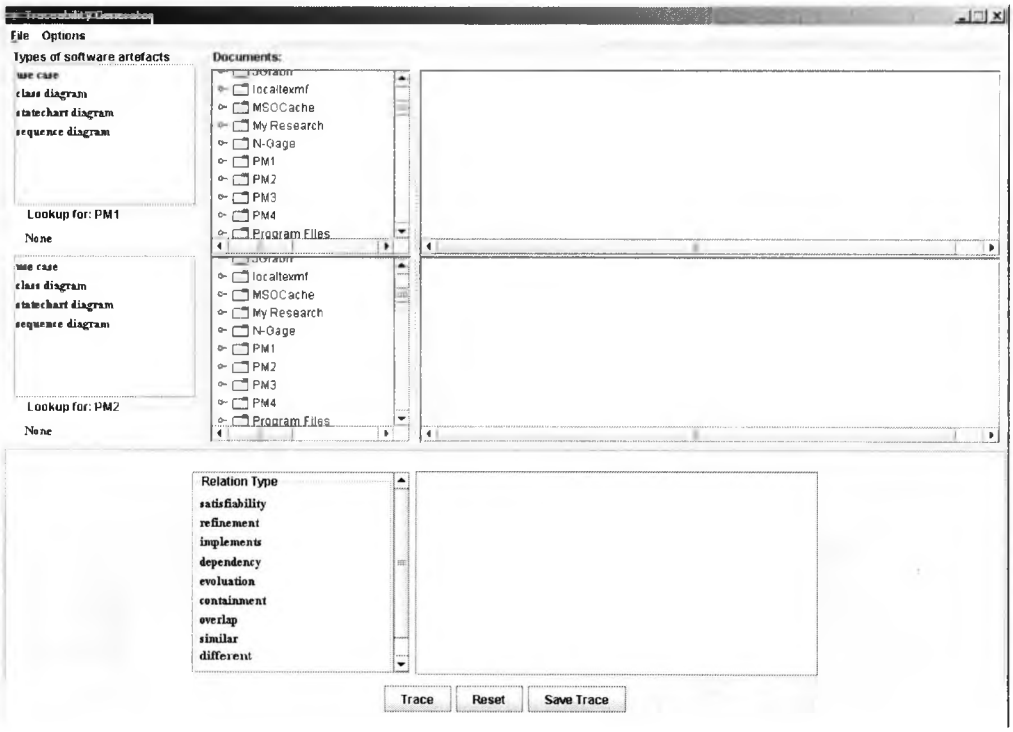


Figure 6- 8: An XTraQue interface for specifying particular documents and relationships according to the specified criteria from the previous interface (Figure 6-5)

However, Figure 6-8 shows the case in which the user has selected documents *use case, class diagram, sequence diagram* and *statechart diagram* of product member *PM1* and *use case, class diagram, sequence diagram* and *statechart diagram* for product member *PM2* from the previous interface (see Figure 6-5 and Figure 6-6).

Figure 6-9 shows how to work with the interface.

- Firstly, the user selects a type of document from the list in panel *a*. The tool displays a list of documents according to the specific type in panel *b*.
- Secondly, if the user selects a particular document in the list in panel *b*, the content of the document is then shown in panel *c* and the document is listed in panel *e* (see Figure 6-9).
- Thirdly, the user specifies the types of traceability relations from the list in panel *d*.

- Optionally, the traceability user selects the "Trace All" button to generate the traceability relations (according to the information in panels *d* and *e*), the "Reset" button to start the selection again, or the "Save Trace" button to save the selection.

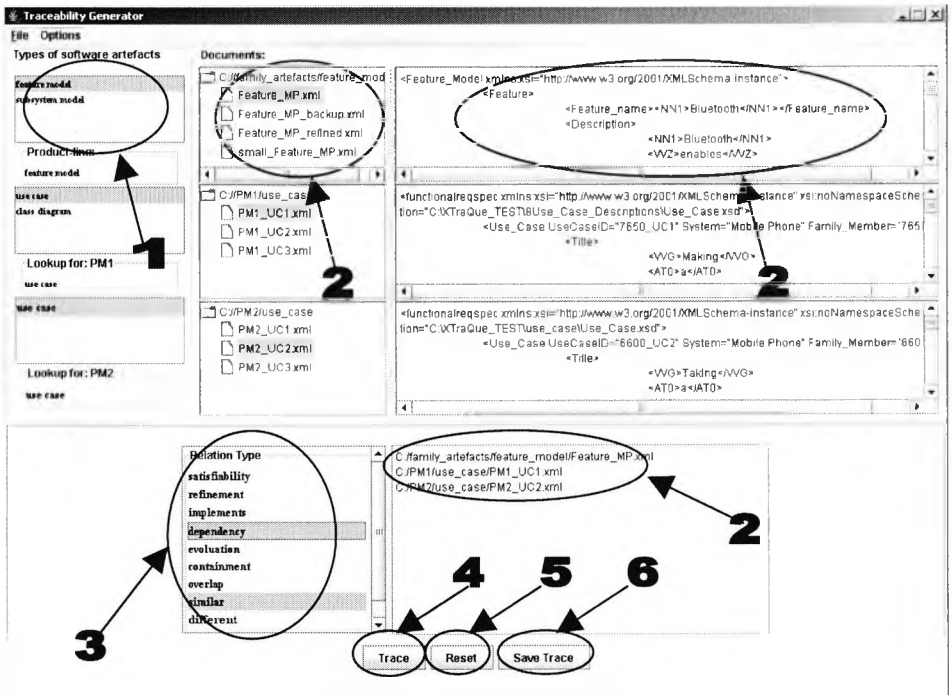


Figure 6- 9: Example interface demonstrating: displaying the context of an XML-based document; and selection of documents types and relationship types to be traced

6.2.4. Editing and Testing XQuery Statements

As shown in Figure 6-10, this interface supports the functionality *Traceability Rule Editor*. This interface consists of:

(a) a panel for editing XQuery statements.

(b) a panel that has six buttons:

- "New" button for resetting the content in panel *a*;
- "Load" button for loading an existing XQuery statement recorded in a file,;
- "Save" button for recording an XQuery statement in panel *a* in a file;

- "Run!" button for execution the XQuery statement in panel *a* and the results are then displayed in panel *c*;
- "Reset" button for resetting the content in panel *c*; and
- "SaveResult" button for recording the results in panel *c* in a file.

(c) a panel for displaying the results of executing the XQuery statement in panel (a).

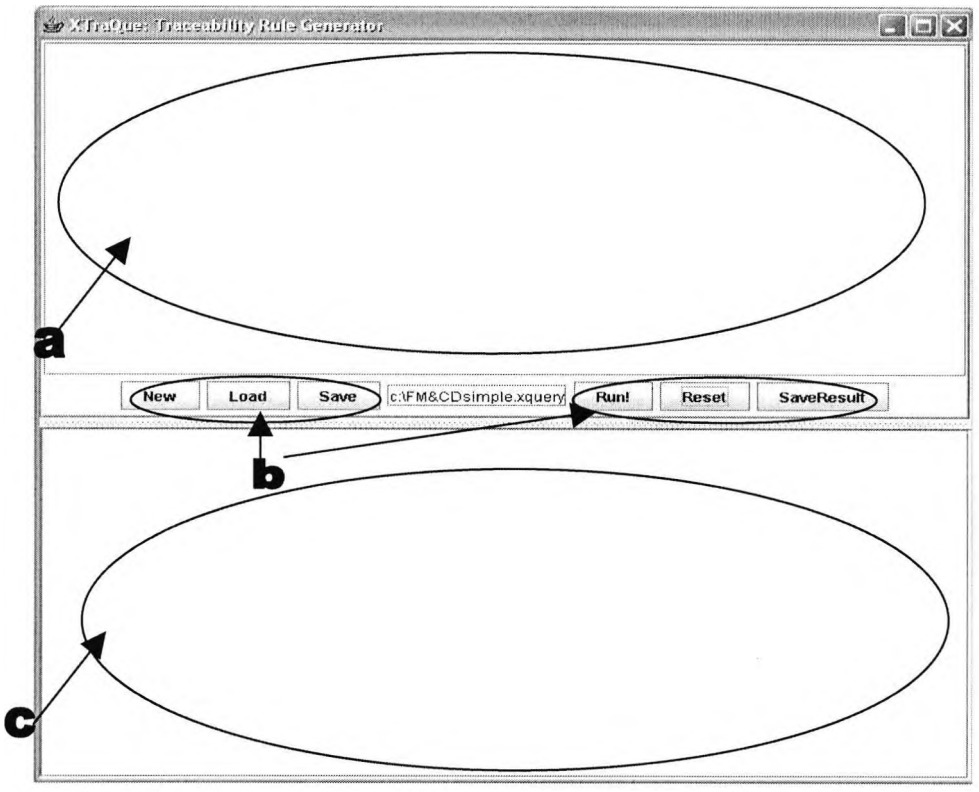


Figure 6- 10: An XTraQue interface for creating and verifying the traceability rules

Figure 6-11 illustrates an example where a user works with the interface.

- Firstly, the user edits XQuery statements in panel *a*.
- Secondly, the user selects an action by clicking a button.
- Finally, the user selects "Run!" button to execute the XQuery statement. The results are shown in the panel *c*.

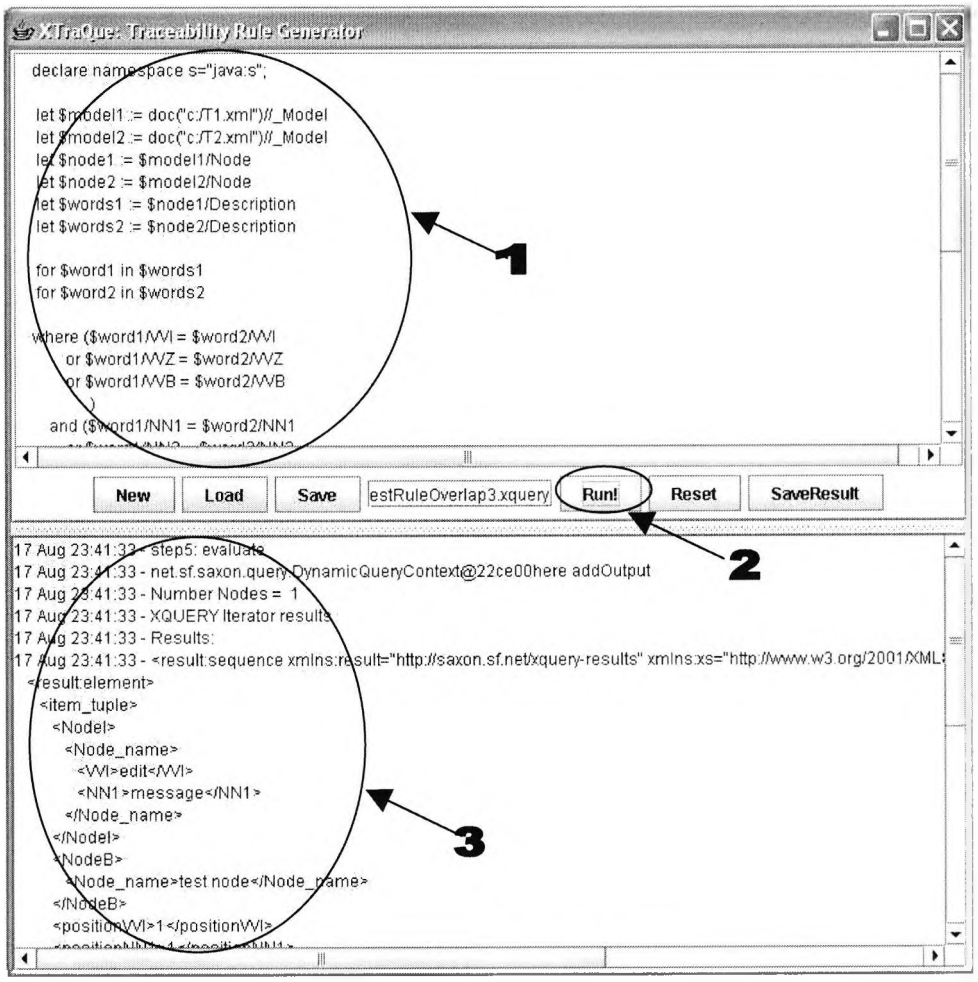


Figure 6- 11: Example interface for creating and verifying traceability rules

6.3. Summary

This chapter has presented the XTraQue tool including its functionalities and user interfaces. The chapter has illustrated the use of the tool to support the automatic generation of traceability relations.

Chapter 7

Mobile Phone Systems – Case Study

This chapter presents the case study of mobile-phone systems. Section 7.1 describes the overview of the case study. Section 7.2 illustrates the documents in the systems. Section 7.3 summarises the chapter.

7.1. Overview of the Case Study

The objectives of the creation of the case study are:

(i) to study the activities during the development of product family systems i.e. domain analysis and domain design for product family, and requirements engineering and design for product members; and

(ii) to demonstrate and evaluate our approach.

The case study has been developed based on study, analysis, and discussions of mobile-phone domain, and ideas in (Nokia) (OMA). Mobile-phone systems are created based on marketing demands which requires a variety of products. In this way, a number of documents are created by stakeholders ranging from market researchers, to requirements engineers, product-line engineers, software analysts, and software developers. Our case study is composed of a family of mobile-phone with three mobile-phone members, namely *PM1*, *PM2*, and *PM3* that we have created. Each member has shared and specialized functionalities with the family. The product members are aimed to satisfy different targets of customers e.g. trendy, luxurious, budget, and high-technology customers.

The list of functionalities and specifications of the mobile-phone members in our case study are shown in Table 7-1 and Table 7-2, respectively. We describe below the details of each mobile-phone product member.

PM1

The product member PM1 is expected to be a trendy phone and targeted for young people. As shown in Table 7-1 and Table 7-2, the product member PM1 has some basic functionalities such as *make and receive calls using GSM 900 and 1800* and *send and receive text messages*. It has also advanced functionalities such as (a) *send and receive multimedia messages* which supports a user to store, download and send pictures and voice with a message, (b) *access Internet* which allows a user to browse and download data based on WAP 1.2.1 technology, (c) *email system* which supports the email protocol e.g. SMTP, POP3, and IMAP4, and (d) *support Java technologies* e.g. CLDC and MIPD. Furthermore, the phone model has advanced features such as (i) *take photos using VGA camera*, (ii) *hand-free speaker*, (iii) *connect via Bluetooth*, and (iv) *infrared*.

PM2

The product member PM2 offers an elegant design and is targeted for users who use a mobile phone for assisting business. As shown in Table 7-1 and Table 7-2, the product member PM2 has basic functionalities such as *make and receive calls using GSM 900, 1800 and 1900* and *send and receive text messages*. It also has functionalities such as (a) *send and receive multimedia messages* which supports a user to store, download and send pictures and voice with a message, (b) *access Internet* which allows a user to browse and download data based on WAP 2.0 and XHTML technologies, (c) *email system* which supports the email protocol e.g. SMTP, POP3, and IMAP4, and (d) *support Java technologies* e.g. CLDC, MIPD, Wireless messaging API, and Mobile media API. Moreover, PM2 has extra features such as (i) *take photos using VGA camera with 2x digital zoom*, and (ii) *play RealOne tunes and videos*.

PM3

The target customers of the product member PM3 are users who enjoy extensive games and music. PM3 introduces a new category of phone games. It has extra

devices like mobile game deck, hand-speaker, Bluetooth and USB, and extra plug-in applications like MP3 player, and stereo FM radio. PM 3 offers advanced functionalities such as (a) *send and receive multimedia messages* which supports a user to store, download and send pictures and voice with a message, (b) *access Internet* which allows a user to browse and download data by supporting WAP 1.2.1 and XHTML technologies, (c) *email system* which supports the email protocol e.g. SMTP, POP3, and IMAP4, and (d) *supporting Java technologies* e.g. CLDC, MIPD, Wireless messaging API, and Mobile media API.

Table 7- 1: Functionalities of Mobile Phone Members

Functionality	PM1	PM2	PM3
F1: Make and receive calls using GSM 900	X	X	X
F2: Make and receive calls using GSM 1800	X	X	X
F3: Make and receive calls using GSM 1900		X	X
F4: Hold and swap a call	X	X	X
F5: Receive and update voice mail	X	X	X
F6: Display and update time and date	X	X	X
F7: Set alarm and time	X	X	X
F8: Record, display, and manipulate call logs	X	X	X
F9: Play games	X	X	X
F10: Update calendar	X	X	X
F11: Add, delete, and update preferences	X	X	X
F12: Add, delete, and update contacts	X	X	X
F13: Include calculator	X	X	X
F14: Take photos using VGA camera	X		
F15: Take photos using VGA camera with 2x digital zoom		X	
F16: FM radio			X
F17: Email system using SMTP, POP3, or IMPA4	X	X	X
F18: Hand-free speaker	X		X
F19: Send and receive text messages	X	X	X
F20: Send and receive multimedia message	X	X	X
F21: Play RealOne format tunes and video		X	
F22: Play and record MP3 format tunes			X
F23: Record and update video (clips)		X	
F24: Play 3GPP video format		X	X
F25: Play Real Video format		X	
F26: Access Internet using WAP 1.2.1	X		X
F27: Access Internet using WAP 2.0		X	
F28: Access Internet using WAP XHTML		X	X
F29: Connect via Bluetooth transfer data	X	X	X
F30: Connect via Infrared transfer data	X	X	
F31: Connect via USB			X
F32: Play MIDI formatted tunes	X	X	X
F33: Play AMR formatted tunes	X		X
F34: Play AAC formatted tunes			X
F35: Play MP3 formatted tunes			X
F36: Play WAV formatted tunes			X
F37: Play True Tones formatted tunes		X	
F38: Compose and play MIDI formatted ring tones		X	X
F39: Record and update voice messages	X	X	X
F40: Transfer data via SyncML and TCP/IP	X	X	X
F41: Support CLDC Java technology	X	X	X
F42: Support MIPD Java technology	X	X	X
F43: Support Wireless messaging API Java technology		X	X
F44: Support Mobile media API Java technology		X	X

Table 7- 2: Specifications of Mobile Phone Members

Specifications	PM1	PM2	PM3
Size	• Weight: 154 g • Dimensions: 114 x 56 x 26 mm, 138 cc	• Weight: 122 g (with BL-5C battery) • Dimensions: 108.6 x 58.2 x 23.7mm, 113cc	• Weight: 137 g • Dimensions: 134 x 70 x 20 mm
Display and User Interface	• Illuminated high-contrast, full-graphics colour display • Graphical user interface • Joystick with five-way navigation	• Bright active matrix TFT colour display • 65,536 colours • 176 x 208 pixels • Graphical user interface with selectable themes • 5-way joystick navigation	• TFT, 4096 colors • Screen size: 176 x 208 pixels • Five-way directional controller
Integrated VGA digital camera	• Image capture at 640 x 480 resolution	• 640 x 480 pixel resolution; standard, portrait, and night modes; 2x digital zoom; self-timer	N/A
Video Recorder	N/A	• Video Recorder: Select picture size QCIF (176x144) or subQCIF (128x96); audio on/off; 2x digital zoom	N/A

Specifications	PM1	PM2	PM3
RealOne Player	N/A	• RealOne Player: Playback and stream RealMedia and 3GPP-compliant content	N/A
Memory Functions	• Phonebook: Up to 500 names • SMS: Up to 100 messages of text • MMS: Up to 50 messages of multimedia • Calendar notes: 100-250 notes (depending on the length of the notes • To do list: Up to 30 notes • 3.6 MB internal shared memory	• Phonebook: Up to 500 names • SMS: Up to 100 messages of text • MMS: Up to 50 messages of multimedia • Calendar notes: 100-250 notes • To do list: Up to 30 notes • 6 MB internal shared memory • Memory card slot for additional user memory.	• Phonebook: Up to 800 names • SMS • MMS: Up to 65 messages of multimedia • Calendar notes: 100 • To-do list: • 3.4 MB internal shared memory • Memory card slot for additional user memory
Messaging	• Text/Multimedia messaging: combine picture, text, voice clip • Email protocols: SMTP, POP3, and IMAP4	• Text/Multimedia messaging: combine image, video, text and voice clip • Email over GSM data, HSCSD, and GPRS • Email protocols: SMTP, POP3, and IMAP4	• Text/Multimedia messaging: Combine image, video, text and voice clip • Email over GSM data, HSCSD, and GPRS • Email protocols: SMTP, POP3, and IMAP4
Wireless Connectivity	• Infrared • Bluetooth	• Bluetooth • Infrared • USB	• Bluetooth

Specifications	PM1	PM2	PM3
Browsing	• Browser supporting WAP 1.2.1 over GSM, HSCSD, and GPRS.	• Browser supporting WAP 2.0 over GSM, HSCSD, and GPRS • Advanced XHTML browser	• Browser supporting WAP 2.0 over GSM, HSCSD, and GPRS • Advanced XHTML browser
Data Transfer	• Up to 43.2 kilobits per second in high-speed circuit switched data networks • Up to 40.2 kilobits per second in GPRS networks	• Up to 40.2Kbps in GPRS networks • Up to 43.2Kbps in HSCSD networks	• Up to 43.2Kbps in GPRS networks • Up to 43.2Kbps in HSCSD networks
Call Management	• Contacts: Advanced contacts database with support for multiple phone and email details per entry, also supports thumbnail picture and groups • Speed dialling • Logs: Keeps lists of your dialled, received, and missed calls • Automatic redial • Automatic answer (works with compatible headset or car kit only) • Supports Fixed Dialling Number, which allows calls only to predefined numbers • Conference call	• Contacts: Advanced contacts database with support for multiple phone and email details per entry, also supports thumbnail picture and groups • Speed dialling • Logs: Keeps lists of your dialled, received, and missed calls • Automatic redial • Automatic answer (works with compatible headset or car kit only) • Supports Fixed Dialling Number, which allows calls only to predefined numbers • Conference call	• Contacts: Advanced contacts database with support for multiple phone and email details per entry, also supports thumbnail picture and groups • Speed dialling • Logs: Keeps lists of your dialled, received, and missed calls • Automatic redial • Automatic answer (works with compatible headset or car kit only) • Supports Fixed Dialling Number, which allows calls only to predefined numbers • Conference call
Java™ Applications	• Supporting Java™ MIDP 2.0 applications	• Supporting Java™ MIDP 2.0 applications	• Supporting Java™ MIDP 2.0 applications

Specifications	PM1	PM2	PM3
Voice Features	• Voice recorder • Integrated handsfree speaker	• Voice recorder	• Voice recorder • Integrated handsfree speaker
Operation	• GSM 900, GSM 1800	• GSM 900, GSM 1800, GSM1900 networks	• GSM 900, GSM 1800, GSM1900 networks
Operation System	• Symbian OS	• Symbian OS	• Symbian OS
Power Management	• Battery Cell BLB-2 830 mAh Li-Ion	• Standard battery (BL-5C) 850 mAh,Li-Ion	• Standard, Li-Ion 850 mAh (BL-5C)

7.2. Documents in the Mobile-Phone Systems

The following sections discuss the documents created in the case study, according to the traceability reference model in Section 4.2. There is a single instance of the feature and subsystem models, but there exist various instances of the process and module models, as well as there exist many instances of use cases, class, statechart, and sequence diagrams. Some examples of these documents in XML format are shown in Appendix D. The complete set of the documents for the mobile-phone family and its three product members can be found at (XTraQue).

7.2.1. Feature Model of Mobile-Phone Systems

As illustrated in Figure 4-1, the feature model in the case study of mobile-phone system has 129 features which are *mandatory*, representing common features, *alternative* and *optional*, representing different features between product members.

For example, all product members must provide *making a call, receiving a call, screen server, wallpaper, and game, calendar*, and *clock* features. However, the product members can support different *network* feature such as CSD, GPRS, GSM, HSCSD, and EDGE. Furthermore, the associations between features are analyzed and captured in the feature model e.g. the product member which provides a *browser* feature has also WAP or XHTML features.

7.2.2. Subsystem Model of Mobile-Phone Systems

We designed five subsystems for mobile-phone systems as shown in Figure 4-5. The brief descriptions of each subsystem are listed as follows:

I. Operating Subsystem

This subsystem provides facilities for performing basic tasks in the mobile-phone systems. Examples of these tasks are: (a) controlling the interaction with all devices, software, and data; (b) performing the interaction between internal applications (e.g.

games, multimedia, and PC connective); (c) responding to internal hardware (e.g. screen, keypad, and Bluetooth), different types of input data (e.g. air signal, keystroke, screen touch, voice) and different types of output data (e.g. air signal, screen-display, voice).

II. Messaging Subsystem

This subsystem manages the exchange and manipulation of messages. It supports two services: short message service (SMS) for textual messages, and multimedia message service (MMS) for multimedia messages. The services are based on a store and forward protocol. The subsystem interacts with short message service centers (SMSC) or multimedia message service centers (MMSC) to receive an incoming message and to forward an outgoing message.

III. Mobile Internet Subsystem

This subsystem manages the interaction between wireless networks and tools such as plug-in applications (e.g. for online games and for mobile browser) and extra hardware (e.g. mobile game desk and 3G PCMCIA data card) for supporting mobile internet applications. The subsystem supports some special functionalities e.g. editing and browsing mobile web pages by using WML and XHTML techniques. The subsystem is also able to activate 24-hour connectivity and support mobile functions e.g. playing online games, managing personal online data, entertaining (playing online radio and video), and servicing online banking.

IV. Network Subsystem

This subsystem supports the communication between different network protocols and maintenance of the network coverage of the mobile-phone devices. It manages a network protocol for passing data over a mobile phone network e.g. GSM, GPRS, HSCSD, CSD and EDGE. Moreover, the subsystem supports different network protocol architectures, for examples, TCP, IPv4, IPv6, MSCHAP v2, IPSec, TCP/IP plug-in framework, WAP stack, and Multiple PDP context.

V. Calling and Applications Subsystem

This subsystem provides the telephony management (e.g. creating and responding phone calls), supports fundamental functions (e.g. a multimode API), and enables the interworking of house-in applications (e.g. electronic games, clock and radio). In particular, the subsystem provides the multimode telephony to enable integrating the rest of the applications interworking and the creation of advanced data services based on global network standards including GSM (Phase 2), GPRS (r4, Class B), CDMA2000 (1x), EDGE (ECSD, EGPRS), and WCDMA (r4).

7.2.3. Process Models of Mobile-Phone Systems

We created two process models i.e. short messaging service (SMS) process model (as shown in Figure 4-7), and Internet application process model (as shown in Figure 7-1). We describe below these process models.

I. Short Messaging Service (SMS) Process model

Short messaging service (SMS) process model is a refinement of the messaging subsystem. The process model illustrates the activities of sending a short text message. The system verifies the network signal, and then interacts with the *short messaging service* (SMS) center. When a phone user has created a short text message, the system sends off the message and waits for a notification. The process model has (a) four resident processes, namely *control, check signal, edit,* and *notification*; (b) one multiple process, namely *short messaging service (SMS) control*; and (c) two transient processes, namely *short messaging service center (SMSC),* and *update remotely*. We describe below each of the above processes:

- *Control* – This process initiates an action of sending a short message when a mobile-phone user has created a short text message and displays an acknowledgement to the user. The process keeps the sent message in the mobile-phone memory.
- *Check signal* – This process performs checking if a signal has been established and is ready for messaging.

- *Update remotely* – This process is to allow update of remote data.
- *Edit* – This process performs the composition of a short message. The short message contains a receiver's address and context. The process provides a list of contacts and a set of template short messages. The process supports two editing modes i.e. alpha mode and predictive mode. The alpha mode accepts alphanumeric. The predictive mode predicts a word from an input keystroke.
- *Short Messaging Service (SMS) Control* – This process performs delivery and receives of a short message to a short message service center (SMSC) that connects the telecommunication network (e.g. GSM, HSCSD, and EDGE) through the short message service gateway mobile switching center (SMS GMSC). This process also attaches extra information about SMSC in a short text message.
- *Short messaging service center (SMSC)* – This process is instantiated by *messaging service centre* (MSC) and responds a message from the SMS control process. The MSC broadcasts the message to the *base station systems* (BSS) and the *base transceiver stations* (BTSs) page the destination MSC.
- *Notification* – This process is to notify incoming messages and acknowledge of sending a short text message.

II. Internet Application Process Model

Internet application (IA) process model is refined for the mobile Internet subsystem. The process model illustrates the activities of accessing the Internet from a mobile-phone set. Initially, the system maintains the reception in order to access the Internet. When the system has received a message from an external process, it enables taking an action i.e. downloading software, restoring data, or launching applications. The process model has five resident processes, namely *trigger, download software, launch application, restore data,* and *maintain receptions,* and one transient process, namely *control.* We describe each of the processes:

- *Trigger* – This process is to notify incoming data to a mobile-phone system.

- *Control* – This process initiates an action of accessing the Internet. The process then interacts with other processes in order to perform downloading software, launching an application, or restoring data. The process keeps the log of transactions in mobile-phone memory.
- *Maintain reception* – This process performs the maintenance of reception between a mobile-phone handset and telecommunication network.
- *Download software* – This process performs downloading software in order to support launching an application on a mobile-phone handset.
- *Launch application* – This process performs launching an application that interacts the Internet.
- *Restore data* – This process performs restoring data in mobile-phone memory.

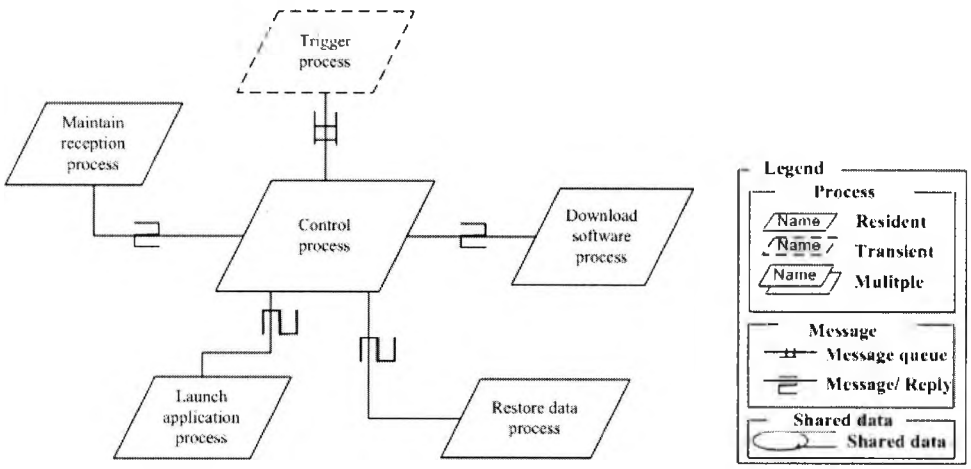


Figure 7- 1: *Internet application* process model

7.2.4. Module Models of Mobile-Phone Systems

We created two module models i.e. the module model (as shown in Figure 4-9) for the process model *short messaging service (SMS) control*, and the module model (as shown in Figure 7-2) for the process model *Internet application.*

I. Module Model for Short Messaging Service (SMS) Control Process Model

This module model illustrates a set of modules and their associations to perform messaging. It contains 18 modules which include (a) 3 *service* modules, (b) 1 *environment handling* module, (c) 10 *technique hiding* modules, and (d) 4 *utility* modules. Table 7-3 presents the description and type of each module.

Table 7- 3: Modules for short messaging service (SMS) control process model

Module	Type	Description
Messaging controller	Precoded	Controlling the messaging.
Connecting	Precoded	Establishing a network communication.
Data controller	Precoded	Controlling internal data of mobile-phone handset.
Multi-network	Precoded	Responding multi-networks.
Signaling controller	Template	Providing algorithms for maintenance the mobile-phone reception and supporting different mobile-phone networks.
IO Interface controller	Precoded	Providing software interfaces for input and output devices of a mobile-phone handset.
Edit controller	Precoded	Managing an editor
Output Interface	Skeleton	Managing output devices of a mobile-phone handset.
Input/ Output Interface	Skeleton	Managing input and output devices of a mobile-phone handset.
Input Interface	Skeleton	Managing input devices of a mobile-phone handset.
Display	Precoded	Displaying data to output devices of a mobile-phone handset.
Touch screen	Precoded	Managing a touch screen of a mobile-

		phone handset.
Keypad	Precoded	Managing a keypad of a mobile-phone handset.
Joystick	Precoded	Managing a joystick of a mobile-phone handset.
Textual display	Precoded	Managing a textual display of a mobile-phone handset to support displaying text.
Web display	Precoded	Managing a graphical display of a mobile-phone handset to support displaying web pages
Timer	Precoded	Setting and displaying time
Data encryption	Precoded	Encrypting and decrypting data.

II. Module Model for Internet Application Process Model

The module model illustrates a set of modules and their associations to perform activities in the Internet application process model. The model contains 22 modules which include (a) 4 *service* modules, (b) 2 *environment handling* modules, (c) 13 *technique hiding* modules, and (d) 3 *utility* modules. Table 7-4 presents the description and type of each module.

Table 7- 4: Modules for Internet application process model

Module	Type	Description
Application controller	Precoded	Controlling a running (local) application.
Connecting	Precoded	Establishing a network communication.
Data controller	Precoded	Controlling internal data of mobile-phone handset.
Mobile-phone Internet application controller	Precoded	Controlling a running Internet application.
Multi-network	Precoded	Responding multi-networks.
Multi-platform	Precoded	Responding multi-platform applications
Signaling controller	Template	Providing algorithms for maintenance the mobile-phone reception and supporting different mobile-phone networks.
IO Interface controller	Precoded	Providing software interfaces for input and output devices of a mobile-phone handset.
WAP controller	Precoded	Controlling WAP browsing
Emailing	Template	Providing algorithms for composing an emails and supporting different emailing protocols.
JavaTM support technique	Template	Managing Java-based plug-ins.
Device Interface	Skeleton	Managing interfaces for extra devices of a mobile-phone handset e.g. game desk, PDA, computers.
Output Interface	Skeleton	Managing output devices of a mobile-phone handset.
Input/ Output Interface	Skeleton	Managing input and output devices

		of a mobile-phone handset.
Input Interface	Skeleton	Managing input devices of a mobile-phone handset.
Display	Precoded	Displaying data to output devices of a mobile-phone handset.
Touch screen	Precoded	Managing a touch screen of a mobile-phone handset.
Keypad	Precoded	Managing a keypad of a mobile-phone handset.
Joystick	Precoded	Managing a joystick of a mobile-phone handset.
Web display	Precoded	Managing a graphical display of a mobile-phone handset to support displaying web pages
Timer	Precoded	Setting and displaying time
Data encryption	Precoded	Encrypting and decrypting data.

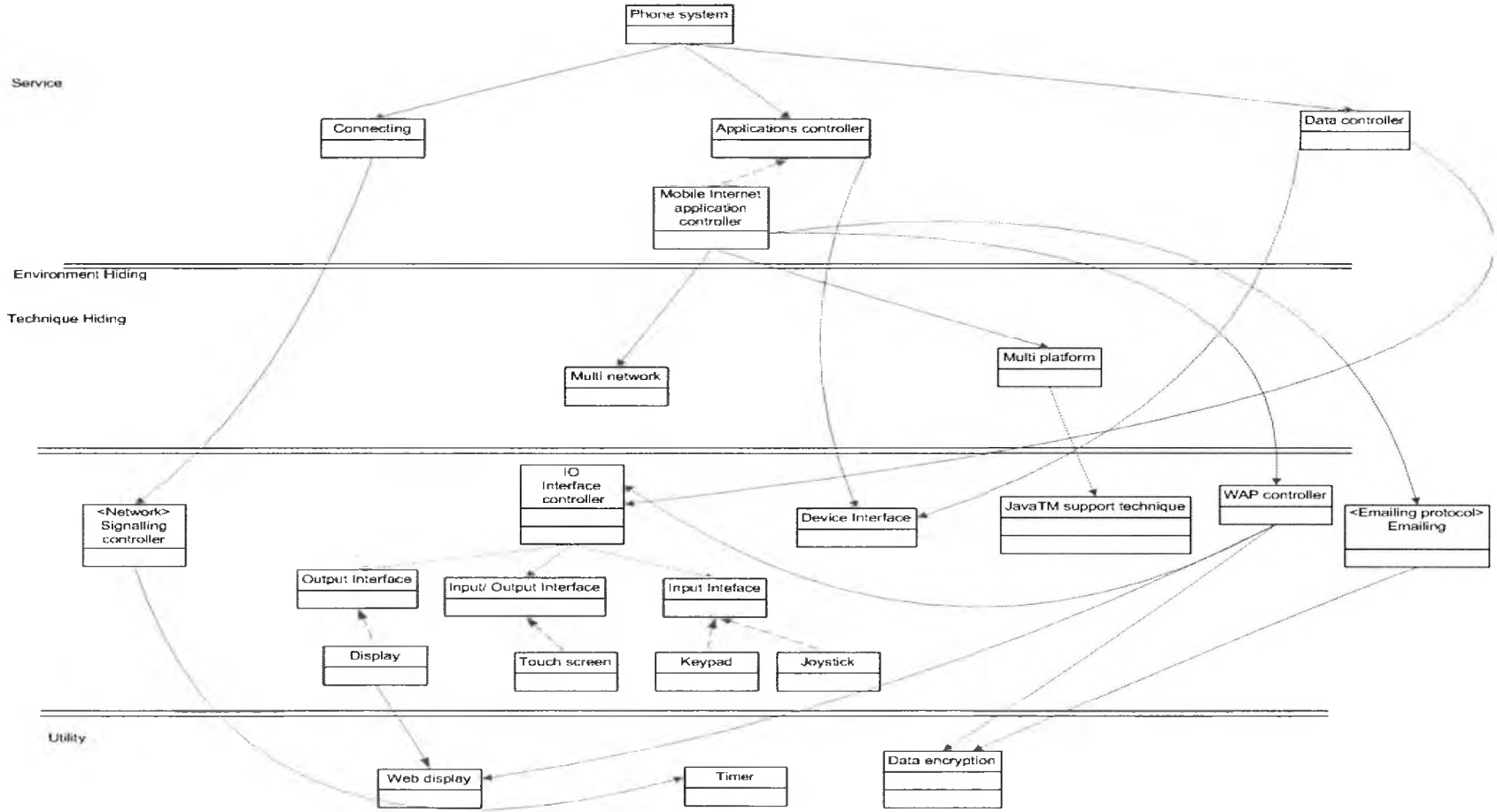


Figure 7- 2: Module model for *Internet application* process model

7.2.5. Use Cases, Class, Statechart, and Sequence Diagrams of Mobile-Phone Members

The use cases are used to elaborate the satisfaction of the functionalities for each product member. We have created four use cases for product member PM1, and four use cases for product member PM2. The four use cases for product member PM1 are: (i) *sending a message*, (ii) *making a call*, (iii) *taking a picture*, and (iv) *sending emails*. The four use cases for product member PM2 are: (i) *making a call*, (ii) *taking a photo*, (iii) *sending emails,* and (iv) *transmitting messages.*

Moreover, we have created:

(a) a class diagram for each product member PM1, PM2, and PM3. Figure 4-11 shows an extract of the class diagram of product member PM3;

(b) a statechart diagram for each product member PM1 and PM2. Figure 4-12 shows a sample statechart diagram of product member PM2; and

(c) four sequence diagrams for product member PM1, and four sequence diagrams for product member PM2. Figure 4-13 shows an extract of a sequence diagram of product member PM2.

7.3. Summary

This chapter has illustrated an overview of the mobile-phone system case study and details of mobile-phone family and its members. The documents are created according to the traceability reference model presented in Section 4.2. and used for demonstration and evaluation our approach that will be presented in Chapter 8.

Part III: Evaluation and Conclusion

Chapter 8

Evaluation and Analysis

In this section, we evaluate and analyze our work. Section 8.1 describes an overview of our evaluation, the different scenarios used to evaluate our work, and an outline of how the evaluation was conducted. Section 8.2 presents the results of the evaluation and analyze these results. Section 8.3 summarises the chapter.

8.1. Evaluation Overview

Our work has been evaluated in order to demonstrate the hypothesis described in Chapter 1 that the work can support

Automatic generation of traceability relations for product family systems

In this evaluation, we have conducted five sets of experiments related to five different scenarios of product family system development. The objectives of these experiments were to evaluate:

(a) how effective XTraQue tool is able to identify relevant documents and files, apply the various traceability rule templates, and create instantiated traceability rules from the templates; and
(b) how effective XTraQue tool is able to generate the traceability relations automatically.

For objective (a), the evaluation was conducted by comparing the number of expected and applied documents, numbers of expected and applied files, numbers of expected and applied traceability rule templates, and numbers of expected and applied instantiated rules. For the case of objective (b), we have measured the *precision* and *recall* of the relevant traceability relations generated by XTraQue.

We have used the following standard definition of recall and precision given in (Faloutsos and Oard. 1995). The authors described that precision measure represents the proportion of retrieved documents which are relevant and recall measure represents the proportion of relevant documents retrieved. More specifically, we applied the precision measure to represent the proportion of generated traceability relations which are valid and recall measure to represent the proportion of valid traceability relations which are generated. The use of precision and recall measurements in traceability literature is also found in (Antoniol et al. 2002; Marcus and Meletic 2003; Hayes et al. 2004; Spanoudakis et al. 2004; Cleland-Huang et al. 2005b). As the following, the precision and recall are calculated by:

$$\text{Precision} = | ST \cap UT | \; / \; | ST |$$

$$\text{Recall} = | ST \cap UT | \; / \; | UT |$$

where

- ST is the set of traceability relations detected by XTraQue;
- UT is the set of traceability relations which are identified by the traceability user, and
- $| X |$ denotes the cardinality of the set X.

The use of recall and precision measures to evaluate traceability approaches have been advocated in (Antoniol et al. 2002, Cleland-Huang et al. 2005b, Hayes et al. 2004, Spanoudakis et al. 2004). Moreover, recall and precision measures are considered common measures for quality results of traceability relation generation.

The scenarios used in our evaluation were based on two main factors. **The first factor** was concerned with the different ways in which organizations can develop product family systems. As proposed in (Krueger 2001) and described in Section 3.3, organisations can develop product family systems in three different ways:

(a) when an organisation decides to analyze, design, and implement a line of products prior to the creation of individual product members (*proactive approach*);
(b) when an organisation enlarges the product family systems in an incremental way based on the demand for new product members or new requirements for existing products (*reactive approach*); and
(c) when an organisation creates a product family based on existing product members by identifying and using common and variable aspects of these products (*extractive approach*).

These approaches are not mutually exclusive and can be used in combination. For instance, it is possible to have product family systems initially created in an extractive way to be incrementally enlarged over time by using a reactive approach.

The second factor was concerned with the stakeholders involved in the product family system development process. Various stakeholders may be involved in this process ranging from market researchers, to product managers, requirement engineers, product-line engineers, software analysts, and software developers. These stakeholders contribute in different ways to the product family system development process, have distinct perspectives of the system, and have distinct interests in different aspects of the product family systems. For example, a market researcher may be interested in the requirements and features of a new product member to be developed, while a software developer may be interested in the design and implementation aspects of this new product member. Therefore, the stakeholders would be interested in different types of documents and traceability relations associated with these documents that may assist them in their various tasks during system development.

In order to take into consideration the various ways of developing product family systems, the heterogeneity of stakeholders, documents, and traceability relation types. The five scenarios used in our experiments include:

(a) the creation of a new product member from existing product family;
(b) the creation of product family from already existing products;
(c) changes to a product member in a product family;
(d) changes to the core assets of a product family; and
(e) impact of changes to the core assets of a product family to a product member.

For each of these scenarios we have identified the stakeholders involved in the process and the types of documents and traceability relations according to the traceability reference model (see Chapter 4) that are related to the scenarios. We describe these five scenarios in the following subsections.

8.1.1. Scenario 1: The creation of a new product member from existing product family

This situation occurs when an organisation wants to enlarge its system and creates a new product member. In this case, traceability relations can be used to support the evolution of software systems and reuse of existing parts of the system. As shown in Figure 8-1, the stakeholders involved in this scenario are:

(a) market researchers that are responsible to identify the feasibility of producing a new product and the features that this new product should include from a commercial point-of-view;
(b) requirements engineers and product managers that specify the requirements of the new product;
(c) product line engineers, product managers, and software analysts that identify which aspects in the core assets of a product family are related to the new product;

(d) software analysts and software developers that analyse existing product members and identify the commonality and differences between existing product members and the new product; and

(e) software developers that design the new product by reusing parts of existing product members and specifying new aspects of the product being developed.

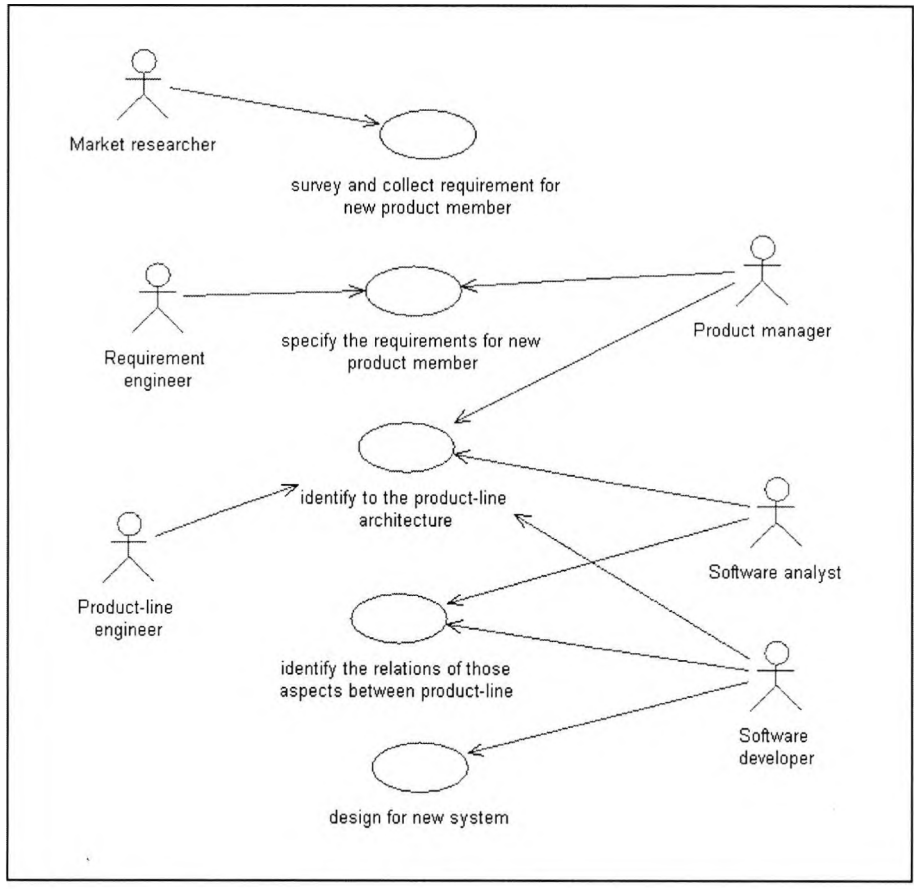


Figure 8- 1: Scenario 1

For this scenario, supposed the situation in which the product family systems contain product member PM2 and the new product member to be developed is PM1 from our case study (see Chapter 7). As shown in Chapter 7, consider that the requirements of PM1 have been specified in four different use cases. In order to be able to identify the similarities and differences between PM1 and PM2, the parts of

PM1 that can be reused from PM2, and the parts of PM1 that need to be developed, it is necessary to compare various documents including feature model of a product family, use cases of PM1 and PM2, and class, sequence, and statechart diagrams of PM2.

The types of documents to be compared and the relevant traceability relations associated with these documents for this scenario are shown in Table 8-1. As presented in the table, the direction of a relation is represented from a row [i] to a column [j] and bi-directional relations appear in two correspondent cells for that relation[10]. The set of use cases of PM1 and PM2 need to be compared with the feature model of a product family in order to support the identification of similarities and differences between use cases of PM1 and PM2. In addition, all class, sequence, and statechart diagrams of PM2 are compared with the use cases of PM1 to assist with the identification of which elements of PM2 design models can be reused. It is also necessary to compare all class, sequence, and statechart diagrams of PM2 with the use cases of PM2 to assist with the identification of similarities and differences between use cases of PM1 and PM2. Moreover, the class, sequence, and statechart diagrams of PM2 need to be compared in order to support the identification of the elements that can be reused when designing PM1.

Table 8- 1: Documents and traceability relations for scenario 1

	Feature Model	**Use Case (PM1)**	**Use Case (PM2)**	**Class Diagram (PM2)**	**Sequence Diagram (PM2)**
Use Case (PM1)	*Contains*		*Similar* *Different*		
Use Case (PM2)	*Contains*	*Similar* *Different*			
Class Diagram (PM2)		*Satisfies* *Implements* *Refines*	*Satisfies* *Implements* *Refines*		
Sequence Diagram (PM2)		*Satisfies* *Implements* *Refines*	*Satisfies* *Implements* *Refines*	*Refines* *Contains*	
Statechart Diagram (PM2)		*Satisfies* *Implements* *Refines*	*Satisfies* *Implements* *Refines*	*Contains*	*Refines*

[10] This will also be the case for tables 8-2, 8-3, 8-4, and 8-5.

8.1.2. Scenario 2: The creation of product family from already existing products

In this case, traceability relations can be used to support the identification of variable and common aspects of existing products in order to create a product family. As shown in Figure 8-2, the stakeholders involved in this scenario are:

(a) product managers that identify which aspects of the product members should be part of the product line;
(b) product line engineers, software analysts, and software developers that design the documents at the product line level; and
(c) software analysts and software developers that develop the documents at the product line level.

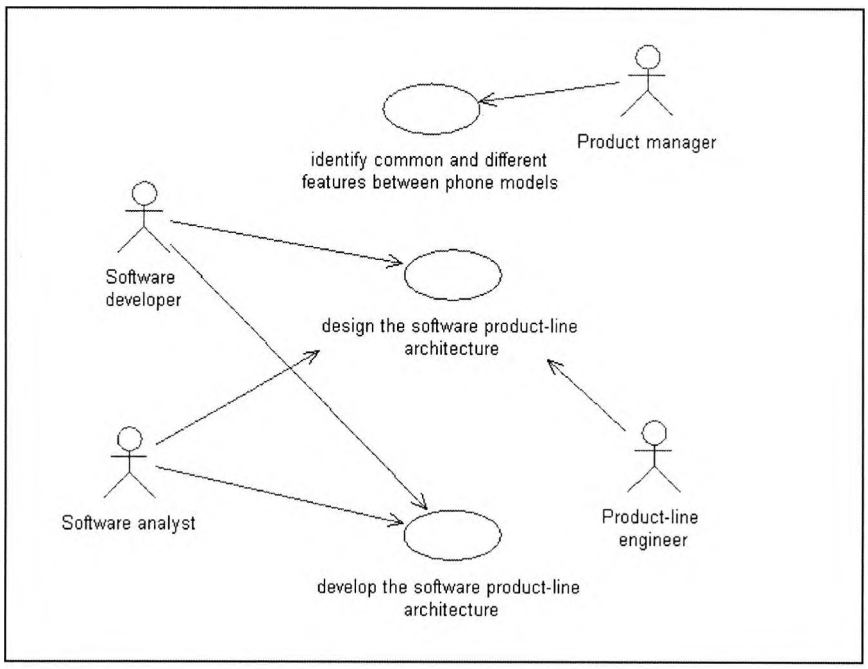


Figure 8- 2: Scenario 2

For this scenario, supposed the situation in which the organization has product members PM1 and PM2 from our case study (see Chapter 7) and would like to create a product family that composes these two members. In this case, all the domain analysis and design models of product members PM1 and PM2 need to be

compared in order to assist with identification of the information that are represented the core assets of the product family. More specifically, similarities and differences between PM1 and PM2 are identified in order to assist the creation of the documents at the product line level (according the traceability reference model in Chapter 4). It is necessary to compare various documents including use cases of PM1 and PM2, and class, sequence, and statechart diagrams of PM1 and PM2.

The types of documents to be compared and the relevant traceability relations associated with these documents for this scenario are shown in Table 8-2. All class, sequence, and statechart diagrams of PM1 and PM2 need to be compared with the user cases of PM1 to assist with the identification of which elements of design models can be deployed for product member PM1. It is also necessary to compare all class, sequence, and statechart diagrams of PM1 and PM2 with the use cases of PM2 to assist with the identification of which elements of design models can be deployed for product member PM2. Moreover, the use cases of PM1 and PM2 need to be compared, and the class, sequence, and statechart diagrams of PM1 and PM2 need to be compared in order to support the identification of which elements are similar and different.

Table 8- 2: Documents and traceability relations for scenario 2

	Use Case (PM1)	Use Case (PM2)	Class Diagram (PM1)	Sequence Diagram (PM1)	Statechart Diagram (PM1)	Class Diagram (PM2)	Sequence Diagram (PM2)	Statechart Diagram (PM2)
Use Case (PM1)		*Similar* *Different*						
Use Case (PM2)	*Similar* *Different*							
Class Diagram (PM1)	*Satisfies* *Implements* *Refines*	*Satisfies* *Implements* *Refines*				*Similar* *Different*		
Sequence Diagram (PM1)	*Satisfies* *Implements* *Refines*	*Satisfies* *Implements* *Refines*	*Refines* *Contains*			*Refines* *Contains*	*Similar* *Different*	
Statechart Diagram (PM1)	*Satisfies* *Implements* *Refines*	*Satisfies* *Implements* *Refines*	*Contains*	*Refines*		*Contains*	*Refines*	*Similar* *Different*
Class Diagram (PM2)	*Satisfies* *Implements* *Refines*	*Satisfies* *Implements* *Refines*	*Similar* *Different*					
Sequence Diagram (PM2)	*Satisfies* *Implements* *Refines*	*Satisfies* *Implements* *Refines*	*Refines* *Contains*	*Similar* *Different*		*Refines* *Contains*		
Statechart Diagram (PM2)	*Satisfies* *Implements* *Refines*	*Satisfies* *Implements* *Refines*	*Contains*	*Refines*	*Similar* *Different*	*Contains*	*Refines*	

8.1.3. Scenario 3: Changes to a product member in a product family

In this scenario traceability relations can be used to support the analysis of the implications of changes in the system. As shown in Figure 8-3, the stakeholders involved in this scenario are:

(a) software analysts that specify changes to be done in a design part of a product member; and
(b) software analysts and software developers that identify the effects of these changes in the other related design software artefacts.

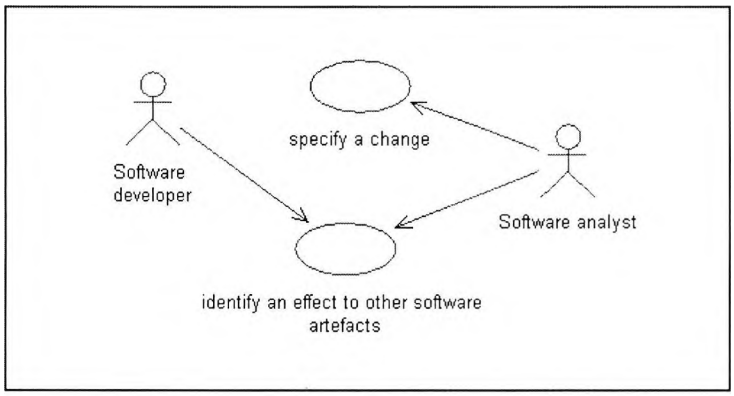


Figure 8- 3: Scenario 3

For this scenario, supposed the situation in which the organisation has developed the core assets of mobile-phone systems with product members PM1 and PM2 from our case study, and that changes are done to product member PM1. Therefore, it is necessary to evaluate how these changes will affect the other design models of PM1 and if these changes also affect the other product members in the product family that may be related to the changes (PM2 in this scenario). The types of documents to be compared and the relevant traceability relations associated with these documents for this scenario are shown in Table 8-3.

Table 8- 3: Documents and traceability relations for scenario 3

	Class Diagram (PM1)	Sequence Diagram (PM1)	Statechart Diagram (PM1)	Class Diagram (PM2)	Sequence Diagram (PM2)	Statechart Diagram (PM2)
Class Diagram (PM1)		*Overlaps*	*Overlaps*	*Similar*	*Overlaps*	*Overlaps*
Sequence Diagram (PM1)	*Depends_on* *Overlaps* *Refines* *Contains*		*Overlaps*	*Depends_on* *Overlaps* *Refines* *Contains*	*Similar*	*Overlaps*
Statechart Diagram (PM1)	*Depends_on* *Overlaps* *Contains*	*Overlaps* *Refines*		*Depends_on* *Overlaps* *Contains*	*Overlaps* *Refines*	*Similar*
Class Diagram (PM2)	*Similar*	*Overlaps*	*Overlaps*		*Overlaps*	*Overlaps*
Sequence Diagram (PM2)	*Depends_on* *Overlaps* *Refines* *Contains*	*Similar*	*Overlaps*	*Depends_on* *Overlaps* *Refines* *Contains*		*Overlaps*
Statechart Diagram (PM2)	*Depends_on* *Overlaps* *Contains*	*Overlaps* *Refines*	*Similar*	*Depends_on* *Overlaps* *Contains*	*Overlaps* *Refines*	

8.1.4. Scenario 4: Changes to the core assets of a product family

In this case, we are interested in investigating how traceability relations can be used to support the evolution and analysis of the impact of the changes to the core assets of a product family. More specifically, this scenario is concerned with changes on the documents at the product line level (according to the traceability reference model in Chapter 4) due to the addition of new features to the product family. As shown in Figure 8-4, the stakeholders involved in this scenario are:

(a) market researchers that identify new features of the system; and

(b) product-line engineers that identify which aspects in the core assets of the product family are related to the new features and the effect of these new features to the other documents at the product line level.

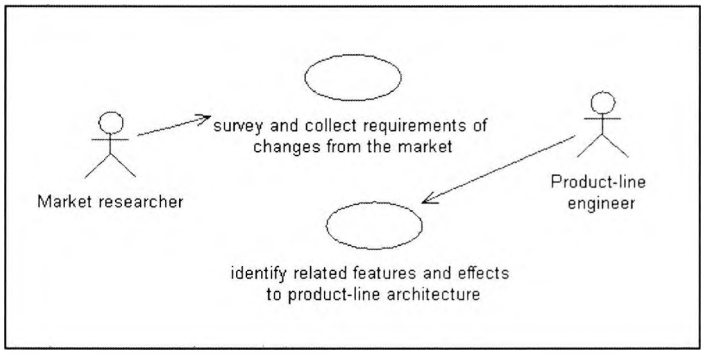


Figure 8- 4: Scenario 4

For this scenario, supposed the situation in which the organisation has developed product family systems from our case study, and that changes are done to the documents at the product line level. Therefore, it is necessary to evaluate how these changes will affect the other documents in the product family. The types of documents to be compared and the relevant traceability relations associated with these documents for this scenario are shown in Table 8-4. In this case, all the documents at the product line level are compared in order to assist with the identification of information that may be affected by change at this level.

Table 8- 4: Documents and traceability relations for scenario 4

	Feature model	**Subsystem model**	**Process model**	**Module model**
Subsystem model	*Satisfies* *Depends_on* *Refines*			
Process model	*Satisfies* *Depends_on* *Refines*	*Refines*		
Module model	*Satisfies* *Depends_on* *Refines*		*Refines*	

8.1.5. Scenario 5: Impact of changes to the core assets of a product family and product members

In this case, we are interested in investigating how traceability relations can be used to support the impact of changes made at the core assets of a product family to product members. This is a small scenario and is concerned with changes at the subsystem of a product family and the impact of these changes in a class diagram. As shown in Figure 8-5, the stakeholders involved in this scenario are:

(a) product line engineers that identify the changes to be done at the subsystem; and

(b) software analysts and developers that identify the effect of these changes at the product member design documents.

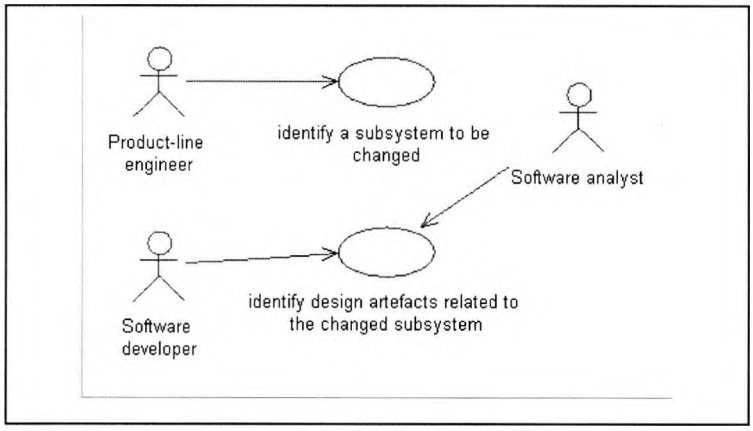


Figure 8- 5: Scenario 5

For this scenario, consider the situation in which we want to analyze the impact of changes at the subsystem model to the class diagram of product member PM3 from our case study. Although changes at the subsystem model can also have impact at other documents at the product member design level (sequence and statechart diagrams), in this experiment we only analyze the relations between subsystem models and class diagram. The types of documents to be compared and the relevant traceability relations for this scenario are shown in Table 8-5.

Table 8- 5: Documents and traceability relations for scenario 5

	Class Diagram
Subsystem Model	*Contains*

8.2. Evaluation Results and Analysis

In this section, we present the results of our evaluation for objectives (a) and (b) described in Section 8.1 and analyze these results.

In the experiments we deployed a total of 63 traceability rule templates that have been instantiated depending on the documents used in each experiment and the traceability relations to be identified. Table 8-6 shows, for each experiment, a summary of the number of documents, number of files, number of (direct and indirect) traceability rule templates, and number of (direct and indirect) instantiated rules that were expected to be used and that have been actually used by the tool. In the table, we consider the number of documents to be different to the number of files since class, sequence, and statechart diagrams of the same product member are represented in the same XMI file due to the nature of XMI.

As seen in Table 8-6, the number of documents, files, traceability rule templates, and instantiated rules expected and the number actually used are the same. In other words, the performance of the XTraQue tool is consistent. The results are consistent across all five scenarios, involving different types of documents, traceability relations, and different sizes of documents and files.

More specifically, in scenarios 1, 2, and 3, the high numbers of instantiated rules were expected to be created; whereas, in scenarios 4 and 5, the low numbers of instantiated rules were expected to be created. This is due to the different types of documents, number of files, and number of traceability relation types used in these scenarios. These results indicate that the performance of identifying documents, files, traceability rule templates and creating instantiated traceability rules by the tool

is consistent across small or large sets of documents and files, and different types of traceability relationships.

Table 8- 6: Summary of documents, files, traceability rule templates, and instantiated traceability rules used in the experiments

	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5
No. of *expected* documents	15	20	12	6	2
No. of *applied* documents	15	20	12	6	2
No. of *expected* files	10	10	2	6	2
No. of *applied* files	10	10	2	6	2
No. of *expected* direct traceability rule templates	17	15	11	7	2
No. of *applied* direct traceability rule templates	17	15	11	7	2
No. of *expected* indirect traceability rule templates	8	11	5	0	0
No. of *applied* indirect traceability rule templates	8	11	5	0	0
Total no. of *expected* traceability rule templates	25	26	16	7	2
Total no. of *applied* traceability rule templates	25	26	16	7	2
No. of *expected* instantiated direct traceability rules	100	192	80	11	2
No. of *actually* instantiated direct traceability rules	100	192	80	11	2
No. of *expected* instantiated indirect traceability rules	8	11	5	0	0
No. of *actually* instantiated indirect traceability rules	8	11	5	0	0
Total no. of *expected* instantiated traceability rules	108	203	85	11	2
Total no. of *actually* instantiated traceability rules	108	203	85	11	2

Tables 8-7 to 8-11 show a summary of the number of traceability relations, for each respective relationship type, manually detected by the user (UT) and automatically detected by XTraQue (ST) in each scenario. As shown in Table 8-7 to 8-11, in each scenario, the numbers of various traceability relations are different. For example, in scenario 1, the tool generated 19 *containment* traceability relations but 172 *implements* traceability relations (see Table 8-7). This indicates that the numbers of traceability

relations being created for each relationship type are not necessarily similar, although they are generated across the same set of documents and files.

Additionally, the numbers of traceability relations for the same type of traceability relationships are not necessarily similar when they are generated in different scenarios involved different documents and files. For example, the tool generated 322 *satisfiability* traceability relations in scenario 2 (see Table 8-8) but 15 *satisfiability* relations in scenario 4 (see Table 8-10).

Moreover, the number of traceability relations being generated depends on the number of relevant documents and files, and traceability relationship types. In scenario 1, 2, and 3, the number of traceability relations are high due to the number of relevant documents and files (see Table 8-6), and many types of traceability relationships (see Sections 8.1.1, 8.1.2, and 8.1.3); while in scenarios 4 and 5, the number of relations are fairly low due to the smaller number of relevant documents and files (see Table 8-6), and less types of traceability relationships (see Section 8.1.4 and 8.1.5). Particularly, in scenario 5 (see Table 8-11), the number of traceability relations being created is very small due to the small number of relevant documents and files, and only one type of traceability relationship being used.

Table 8- 7: Summary of traceability relations detected in scenario 1

Types of traceability relations	UT	ST
No. of *implements* traceability relations identified	166	172
No. of *satisfiability* traceability relations identified	154	154
No. of *containment* traceability relations identified	23	19
No. of *refinement* traceability relations identified	176	180
No. of *similar* traceability relations identified	333	333
No. of *different* traceability relations identified	329	341

Table 8- 8: Summary of traceability relations detected in scenario 2

Types of traceability relations	UT	ST
No. of *implements* traceability relations identified	388	410
No. of *satisfiability* traceability relations identified	324	322
No. of *containment* traceability relations identified	16	16
No. of *refinement* traceability relations identified	348	342
No. of *similar* traceability relations identified	1404	1402
No. of *different* traceability relations identified	8	16

Table 8- 9: Summary of traceability relations detected in scenario 3

Types of traceability relations	UT	ST
No. of *dependency* traceability relations identified	28	28
No. of *overlaps* traceability relations identified	28	28
No. of *containment* traceability relations identified	20	20
No. of *refinement* traceability relations identified	52	60
No. of *similar* traceability relations identified	126	130

Table 8- 10: Summary of traceability relations detected in scenario 4

Types of traceability relations	UT	ST
No. of *satisfiability* traceability relations identified	20	15
No. of *dependency* traceability relations identified	2	2
No. of *refinement* traceability relations identified	4	4

Table 8- 11: Summary of traceability relations detected in scenario 5

Types of traceability relations	UT	ST
No. of *containment* traceability relations identified	6	6

Additionally, Table 8-12 shows, for each scenario, a summary of the number of traceability relations detected and grouped as direct, indirect, and total traceability relations and the number of traceability relations in the intersection of the relations generated by XTraQue tool and manually identified by the user. In the table, ST_{total} is the set of traceability relations detected by XTraQue; ST_{direct} is the set of *direct* traceability relations detected by XTraQue; $ST_{indirect}$ is the set of *indirect* traceability relations detected by XTraQue. UT_{total} is the set of traceability relations identified by traceability user; UT_{direct} is the set of *direct* traceability relations identified by traceability user; and $UT_{indirect}$ is the set of *indirect* traceability relations identified by traceability user.

Table 8- 12: Summary of traceability relations detected in the experiments

	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5
UT_{direct}	519	1076	128	26	6
ST_{direct}	525	1090	136	21	6
$\lvert ST_{direct} \cap UT_{direct} \rvert$	502	1046	112	17	5
$UT_{indirect}$	333	1412	126	0	0
$ST_{indirect}$	341	1418	130	0	0
$\lvert ST_{indirect} \cap UT_{indirect} \rvert$	282	1208	105	0	0
UT_{total}	852	2488	254	26	6
ST_{total}	866	2508	266	21	6
$\lvert ST_{total} \cap UT_{total} \rvert$	784	2254	217	17	5

As shown in Table 8-12, for scenarios 4 and 5, the cells corresponding to the number of indirect traceability relations ($UT_{indirect}$ and $ST_{indirect}$) contains value zero (0) since these scenarios do not involve indirect traceability relations, and not because the tool cannot generate these relations or the use cannot identify these relations manually. Moreover, the high number of traceability relations detected in scenarios 1 and 2 is due to the number of document types and traceability relation types used in these scenarios, as well as the specific documents that are related through the various relation types. For instance, in scenario 1, there are (a) four use case documents for PM1 and four use case documents for PM2 that are related in terms of three different types of traceability relations (satisfies, implements, and refines) with one class diagram, four sequence diagrams, and one statechart diagram; (b) four use case documents for PM1 and four use case documents for PM2 that are related in terms of contains relations with feature model; (c) four use case documents of PM2 that are related to four use case documents of PM1 in terms of similar and different relations; (d) four sequence diagrams of PM2 that are related in terms of refines and contains relation types with one class diagrams and in terms of refines relation with one statechart diagram; and (e) one class diagram that is related in terms of contains relations with one statechart diagram. A similar and more complex situation occurs in scenario 2.

Figure 8-6 shows charts comparing the numbers of traceability relations generated by the user and XTraQue tool in each scenario (as shown in Table 8-12). Each chart

has three columns, namely *A* representing a number of traceability relations generated by both user and XTraQue tool ($|\mathbf{ST}_{total} \cap \mathbf{UT}_{total}|$); *B* number of traceability relations detected by the user ($\mathbf{UT}_{total}$); and *C* number of traceability relations generated by XTraQue tool ($\mathbf{ST}_{total}$).

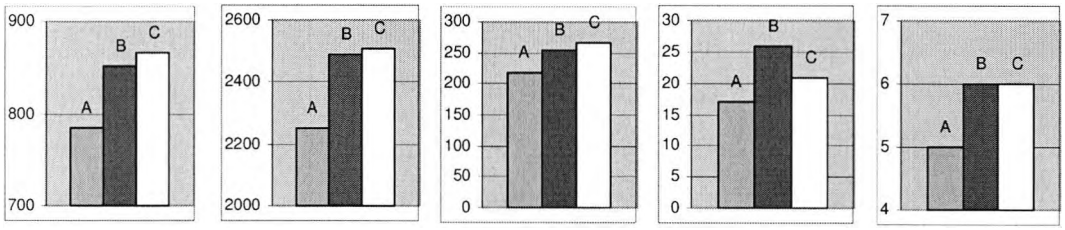


Figure 8- 6: Traceability relations detected by the traceability user and XTraQue (A), by traceability user (B), and by XTraQue (C) in five experiments

Table 8- 13: Precision and Recall Rates (%)

	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Scenario 5	Average
Precision of generating *direct* relations	95.6	95.9	82.3	81.0	83.4	87.6
Precision of generating *indirect* relations	82.7	85.2	80.7	-	-	82.8
Precision of generating *all* relations	90.5	89.8	81.6	81.0	83.4	**85.3**
Recall of generating *direct* relations	96.7	97.2	87.5	65.4	83.4	86.0
Recall of generating *indirect* relations	84.7	85.5	83.4	-	-	84.5
Recall of generating *all* relations	92.0	90.6	85.4	65.4	83.4	**83.3**

Table 8-13 shows the results of our experiments for each scenario in terms of recall and precision rates including the number of direct and indirect traceability relations identified by the users and by the tool. The traceability relations generated by the tool in each different scenario were compared against traceability relations manually identified by users with substantial experience and training in software engineering and product family systems. Additionally, for scenarios 4 and 5, the cells corresponding to the precision and recall rates of generating indirect relations do

not show any values since these scenarios do not involve the generation of indirect relations as mentioned earlier (see Table 8-12).

The results shown in Table 8-13 provide positive evidence about our approach to automatic generate traceability relations at a high level of recall and precision. The results show that direct traceability relations have higher precision and recall values when compared to indirect traceability relations (scenarios 1, 2, and 3). This is due to the fact that indirect traceability relations are generated based on direct traceability relations and in the cases of incorrect direct traceability relations generated by the tool, or missing direct traceability relations by the tool, these will interfere with the precision and recall of the indirect traceability relation. More specifically, the incorrect and missing direct traceability relations will cause a lower precision, while the missing direct traceability relations will contribute to a lower recall. Moreover, scenario 4 has the lowest recall when compared with the other scenarios. We attribute this to the fact that this scenario has the lowest number of direct rule templates with respect to the different types of traceability rules used in the scenario when compared to scenarios 1, 2, and 3 (scenario 5 is quite a small scenario to be considered in this case). For example, scenario 1 has 17 direct rule templates for four different types of direct relations, scenario 2 has 15 rule templates for four different types of direct relations, and scenario 3 has 11 rule templates for four different types of direct relations, while scenario 4 has only seven rule templates for three different types of relations. Therefore, the number of direct traceability relations generated by XTraQue for scenario 4 is smaller than the number of traceability relations identified by the user. In the other scenarios we observe an inversion of this situation (i.e., number of direct traceability relations generated by the tool is higher than the ones generated by the users). The addition of new traceability rules for satisfiability, dependency, and refinement relations for documents at the product line level cause an increase in the recall.

We applied the bar chart to compare the precision and recall in the experiments. Figure 8-7 (a) and Figure 8-7 (b) show that the precision figures in all the scenarios and the recall figures in all the scenarios are not so significant. On average, the

performance of our approach in terms of precision and recall measurements in five scenarios seems to be consistent (see Figure 8-7 (a)). Particularly high are the precision values in all five scenarios (ranging from 81.0% to 95%) and the recall values in scenario 1, 2, 3, and 5 (ranging from 83.4% to 92.0%). Although the recall value in scenario 4 is slightly lower than the others in different scenarios, the value for 65.4% of recall is still comparable to the average recall value achieved in (Spanoudakis et al. 2004) and higher than the average recall value achieved in (Antoniol et al. 2002).

The charts in Figure 8-7 (b) show that the precision values from five scenarios are ranging from 81.0% to 90.5% and the recall values are ranging from 65.4% to 92.0%. These rates indicate that the precision values and recall values achieved by our approach are fairly consistent, although the generation of traceability relations were performed between various numbers of documents, files, and different types of traceability relationships.

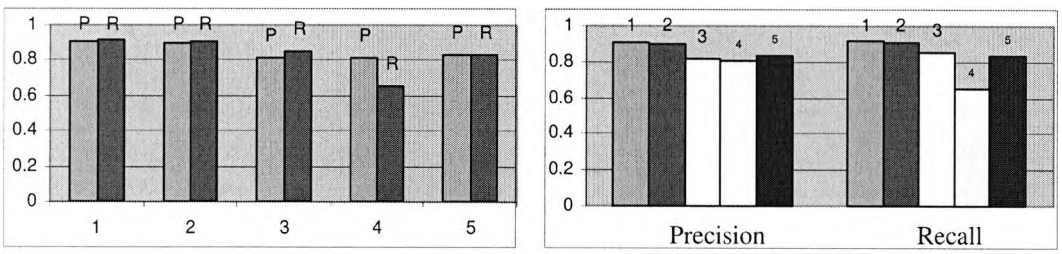


Figure 8- 7: (a) Precision and recall figures of each scenario; (b) Comparison of precision and recall figures from five scenarios

Overall, the average precision measured, 85.3%, and average recall measured 83.3%, in our experiments are encouraging results. According to Table 8-14, it shows the summary of recall and precision measurements achieved by several existing traceability approaches. In the table, our precision results are better than, and our recall results are comparable to, the results achieved in the approach (Antoniol et al. 2002) that support automatic generation of traceability relations between requirements specifications and source code based on probabilistic and vector space

information retrieval approaches. In their work, they have managed to achieve 61% of recall and 56.5% of precision. The results achieved in this work are also better than the results of the work for automatic generation of traceability relations between textual documents representing requirements, use cases, and analysis models of software systems (Spanoudakis et al. 2004). In this case, the authors have achieved 69.6% of recall and 77.2% of precision. Similarly, the results achieved in our work are also better than the results achieved in (Hayes et al. 2004). The authors applied three vector space IR techniques to enhance the generation of traceability relations and have achieved 39.2% of precision and 80.9% of recall. Although the results achieved in the work (Cleland-Huang et al. 2005b) have achieved 90.2% of recall, they have achieved fairly low 20.6% of precision.

Table 8- 14: Summary of recall and precision rates achieved by several existing traceability approaches

Approach	Average Recall (%)	Average Precision (%)
(Antoniol et al. 2002)	61.0	56.5
(Cleland-Huang et al. 2005b)	90.2	20.6
(Spanoudakis et al. 2004)	69.6	77.2
(Hayes et al. 2004)	80.9	39.2
Our approach	83.3	85.3

The results of our experiments have demonstrated our initial hypotheses and that XTraQue can be effectively used to automatically generate traceability relations for product family systems. Of course, 100% of recall and precision are the ultimate goal of any tool. However, to the best of our knowledge, none of the existing approaches managed to achieve this goal.

Additionally, the time spent during the generation of the traceability relations in the five scenarios used in our evaluation varies depending on the size of the documents and the number of various relationship types. For example, scenarios 1 and 2 took longer to be executed than scenarios 4 and 5 that are significantly smaller. Moreover, the textual characteristics of some of the documents and the number of rules that may exist for a certain relationship type also contributes to an increase of the processing time. However, our XTraQue tool allows a user to select specific

documents and traceability relationship types to be processed in an interaction. This feature of the tool can be used to assist and control the amount of time during traceability generation.

8.3. Summary

This chapter have illustrated the experiments and their results. We have observed the effectiveness of the tool as well as evaluated the results of traceability generation by applying with the precision and recall measurements. In addition to, the explanations of results have been given. The evaluation and analysis leads to the conclusion of this thesis that will be presented in next chapter.

Chapter 9

Conclusions and Future work

We provide in this chapter the conclusions, findings, and future work of this thesis. Section 9.1 presents the overall conclusions. The findings of this thesis and the future work are described in Section 9.2 and Section 9.3, respectively. The final remarks are listed in Section 9.4.

9.1. Overall Conclusions

The research in this thesis has contributed to enable traceability of product family systems in an automatic way. We summarize below the contributions in this thesis.

A traceability reference model for product family systems – In this thesis, we proposed a traceability reference model for product family systems in Chapter 4. The concepts and motivation are derived from the background in Chapters 2 and 3. The model is composed of two main components. Firstly, it includes a set of documents created during the development of product family systems. Our approach applied the *feature oriented reuse method* (FORM) and the *unified modeling language* (UML). Eight types of documents are concerned with these methods, namely: (a) *feature model* used to represent reference requirements of product family systems; (b) *subsystem model*, (c) *process model*, and (d) *module model* used to represent a software product line architecture; (e) *use case* used to represent requirements of product members; and (f) *class diagram,* (g) *sequence diagram*, and (h) *statechart diagram* used to represent design models of product members.

Secondly, the model includes the classification of traceability relationships between these documents. Two groups of traceability relationships are defined, namely: (i)

direct, which are traceability relations being identified straightforwardly between documents; and (ii) *indirect*, which are traceability relations being identified based on the existing direct traceability relations. The direct traceability relations are classified in seven types, namely: (a) *implements* relations holding between two documents that an artefact in a document executes or allows for the achievement of an artefact in another document; (b) *containment* relations holding between two documents that an artefact in a document uses another artefact in another document; (c) *refinement* relations holding between two documents that a document specifies more details about another document; (d) *satisfiability* relations holding between two documents that an artefact in a document meets the expectation and needs of another artefact in another document; (e) *overlap* relations holding between two documents that refer to common aspects of a system; (f) *dependency* relations holding between two documents that an artefact in a document relies on the existence of another artefact in a document; and (g) *evolution* relations holding between two documents that are evolved during the product family system development. Additionally, the indirect traceability relations are classified in two types, namely: (a) *similar* relations holding between two documents of the same type and assisting with the identification of common aspects between various product members; and (b) *different* relations holding between two documents of the same type for different product members and assisting with the identification of variable aspects between various product members.

The rule-based approach for generating the traceability relations – In this thesis, we proposed a rule-based approach for enabling traceability activities in the domain of product family systems in Chapter 5. A set of traceability rules is developed and used to justify the identification of traceability relations automatically. Two groups of traceability rules, namely *direct* and *indirect*, support for the creation of direct and indirect traceability relations, respectively. The rules are expressed in an extension of XQuery that includes extra functions implemented in XQuery and Java languages. The identification of traceability relations is based upon four criteria as follows: (a) semantics of document types, (b) types of traceability relations, (c) part-of-speech of words in textual sentences, and (d) synonyms and

distance of words being compared in a text. When the conditions in a rule are verified, traceability relations are generated and represented in XML documents.

The demonstration and evaluation of the approach – The prototype tool in Chapter 6, called *XTraQue,* is implemented in Java to facilitate the demonstration and evaluation of the approach in Chapter 8. The tool also encompasses Saxon as an XQuery processor. The main functionalities of the tool are namely: (a) *Traceability Selection*, specifying documents to be traced and the types of traceability relationships to be created; (b) *Traceability Creation*, identifying traceability relations according to the criteria from Traceability Selection; (c) *Traceability Presenter*, recording and representing the traceability relations identified by Traceability Creation; and (d) *Traceability Rule Editor*, testing and displaying results of a traceability rule.

Additionally, a case study of mobile-phone systems in Chapter 7 is used for demonstration and evaluation the approach. Five scenarios are created to demonstrate different situations of traceability activities in the product family system development, involving (a) establishing traceability relations between different types of documents; (b) identifying different types of traceability relations; and (c) supporting different stakeholders. The experiments of traceability generation have been evaluated by considering two criteria: (i) identifying relevant documents, identifying traceability rule templates, and creating instantiated traceability rules; and (ii) generating traceability relations. For the latter criteria, the *precise* and *recall* measurements are used.

9.2. The Findings

This thesis has shown that some degree of laborious process in generating traceability relations for a large number of heterogeneous documents can be reduced with the tool support implemented. Traceability activities i.e. generation, recording, and representation of traceability relations are done in an automatic way. The generation of traceability relations captures the semantics that are represented through the traceability relationship types. The relations are identified between, at

least, two documents. Some traceability relations such as indirect relations are created based upon two direct relations. An indirect relation therefore draws associations between three documents. More specifically, *similar* and *different* traceability relations are identified between two documents of the same type whereas the documents have a same relationship type to another document of another type. As shown in this thesis, the results of generation are measured by using *precision* and *recall* rates. The average precision measured as 85.3% and average recall measured as 83.3%. The results achieved by the research in this thesis are better than the results achieved in existing traceability approaches.

Additionally, this thesis has demonstrated different aspects to other existing approaches. We describe below the aspects:

- *Enables traceability of product family systems:* Although some existing approaches (Alexander 2003, Antoniol et al. 2002, Bayer and Widen 2002, Cleland-Huang et al. 2002b, Dick 1999, Egyed 2003, Gotel and Finkelstein 1995, Hayes et al. 2003, Kim et al. 2005, Knethen 2002a, Lago et al. 2004, Letelier 2002, Lindvall and K. 1996, Maletic and Marcus 2001, Marcus and Meletic 2003, Pinheiro and Goguen 1996, Pohl 1996b, Ramesh and Jarke 2001, Sherba et al. 2003a, Toranzo and Castro 1999) suggested traceability activities in the software system development, they do not support traceability of product family systems. Neither the traceability reference models nor classification of traceability relations are proposed for identifying common and variable aspects in the domain of product family systems. In this thesis, we defined the traceability reference model and classification of traceability relations in the domain of product family systems.
- *Enables the automatic generation of traceability relations for the domain of product family systems:* Although some existing approaches (Bayer and Widen 2002, CAFE 2003, Dick 1999, ESAPS, Kim et al. 2005, Riebisch and Philippow 2001, Toranzo and Castro 1999) proposed the traceability

activities in the domain of product family systems, they do not provide tool support or define the process for achieving the activities. In this thesis, we have provided the XTraQue tool for enabling traceability generation, recording and representation in an automatic way. The tool has sophisticated and user-friendly interfaces to facilitate the activities.

- *Generates traceability relations in different levels of granularity:* In this thesis, different relationships are generated between documents created from different activities of the development life cycle for product family systems. Two views of the relationships can be categorized: (a) coarse-grained associations such as traceability relations between different product members, and between core assets and product members; and (b) fine-grained associations such as traceability relations between elements in documents. However, some existing approaches (Kim et al. 2005, Lago et al. 2004, Leite and Breitman, Riebisch and Philippow 2001, Toranzo and Castro 1999) defined the traceability relations in product family systems either for fine-grained or course-grained associations, but not for both.

The research in this thesis has demonstrated the possible situations of the use of traceability relations during the development of product family systems. We describe below the use for its different purposes:

- *Reuse:* The research in this thesis has found that the degree of reusing core assets of product family systems affects the cost of the development of the systems. The cost of the product family system development depends on the proportion of reuse of the core assets for the development of product members. However, the poor reuse would have caused higher cost to the product family system development. Traceability relations are used to assist the development by reducing the cost i.e. effort and time. Since it is a common situation that stakeholders such as software engineers need to relate the existing software artifacts

to new requirements in order to assist the development of the new requirements. The research in this thesis has demonstrated that traceability relations can be used to facilitate such activities in the situation.

- *Understanding:* The research in this thesis has shown that different stakeholders, who have different experiences in the product family system development process, have different perspectives regarding to software artefacts. Traceability relations enable stakeholders to comprehend the associations between the artefacts in an easier way. Coarse-grained associations such as common and variable aspects between product members and fine-grained associations such as ones between elements in different artefacts are illustrated though traceability relations and can facilitate the understanding of the generated documents.

9.3. Future Work

A number of possible directions for further investigations have been identified. We provide in this section future work of the research in this thesis, what needs to be done to improve the approach and to increase the benefits of the approach:

- **Visualisation:** As shown in this thesis, a large number of traceability relations can be generated across product family system documents. It is therefore believed that the approach could be extended and enhanced to support a better way of visualizing the relations in various perspective views. In addition, sophisticated techniques for visualization could support the use of traceability relations more efficiently.
- **Domain Implementation:** The research in this thesis has focused on two main activities of product family system development i.e. analysis and design. The approach could be expanded to cover the activity of

implementation in order to complete the whole life-cycle of the development of product family systems.

- **Document specification:** As shown in this thesis, the activities of traceability start after the creation of the various artefacts. It is therefore believed that the approach could benefit by providing tool support for the specification of documents.
- **Tracing between different product families**: As shown in this thesis, the approach can enable traceability practice in a single product family. However, it is believed that the approach could equally benefit from support traceability between different product families.
- **Reduction of traceability generation time**: As shown in this thesis, the traceability generation can take a long time to be processed depending on the size, number, and types of documents and relationships. More work needs to be done to optimize the processing time.

9.4. Final Remarks

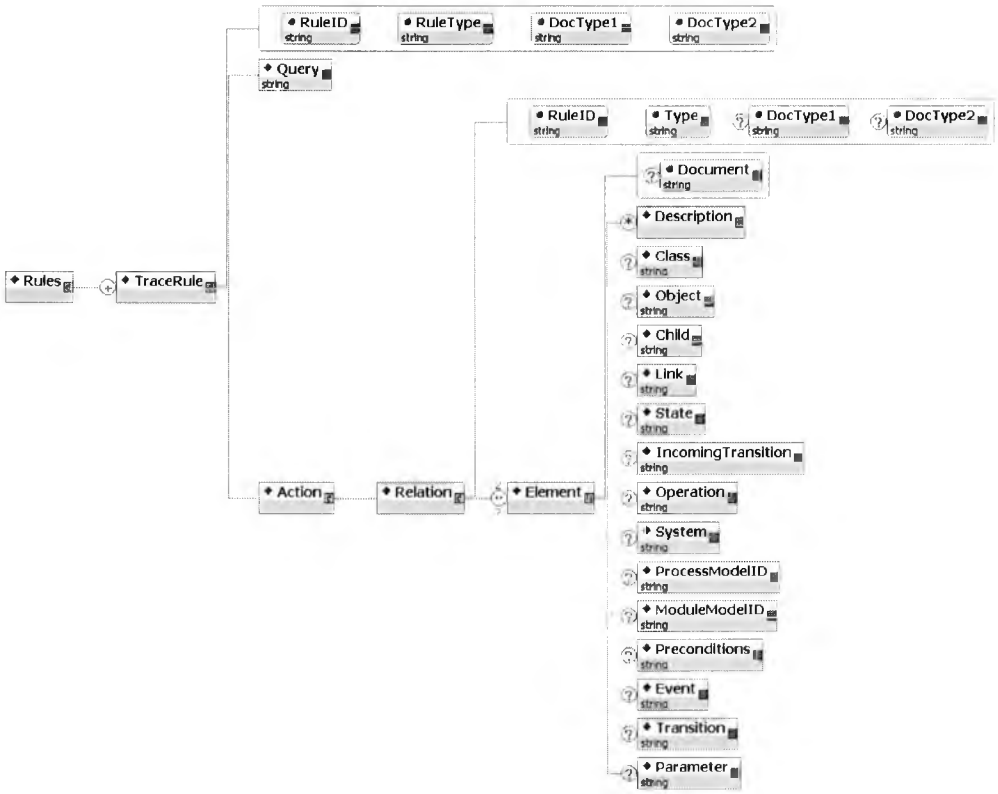
This thesis has presented the rule-based approach for software traceability on product family systems. The research in this thesis has been contributed to:

- provide the background of traceability to software systems (Chapter 2);
- provide the background of product family systems (Chapter 3);
- present the traceability reference model (Chapter 4);
- present the framework of automatic traceability generation process in product family systems (Chapter 5);
- illustrate the XTraQue tool (Chapter 6);
- provide a case study in the domain of mobile-phone systems (Chapter 7); and
- demonstrate and evaluate the approach (Chapter 8).

Appendices

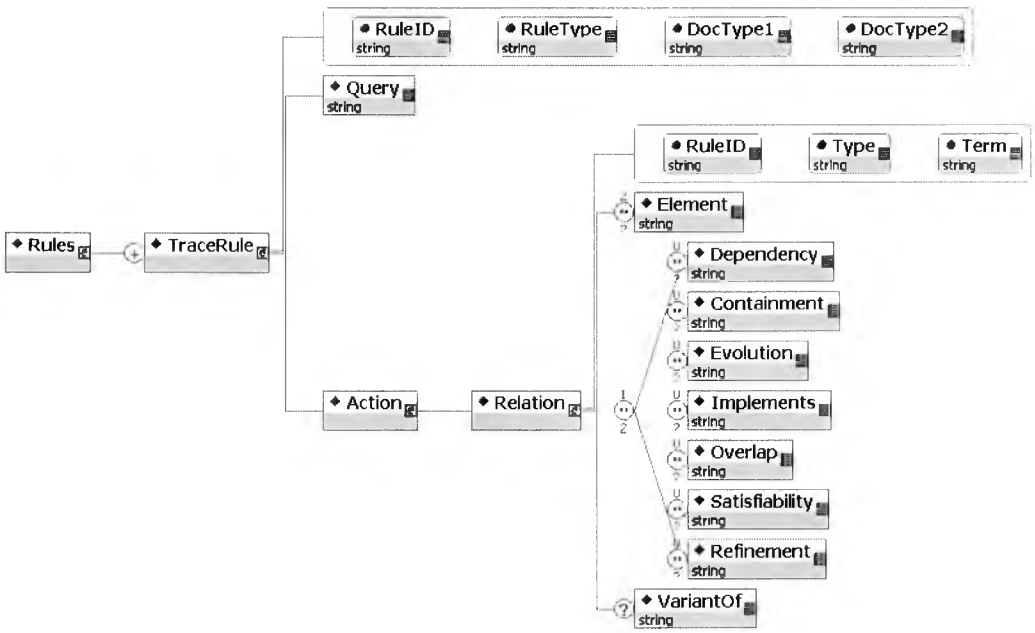
Appendix A – XML Schemas

A.1. XML Schema for direct traceability rules



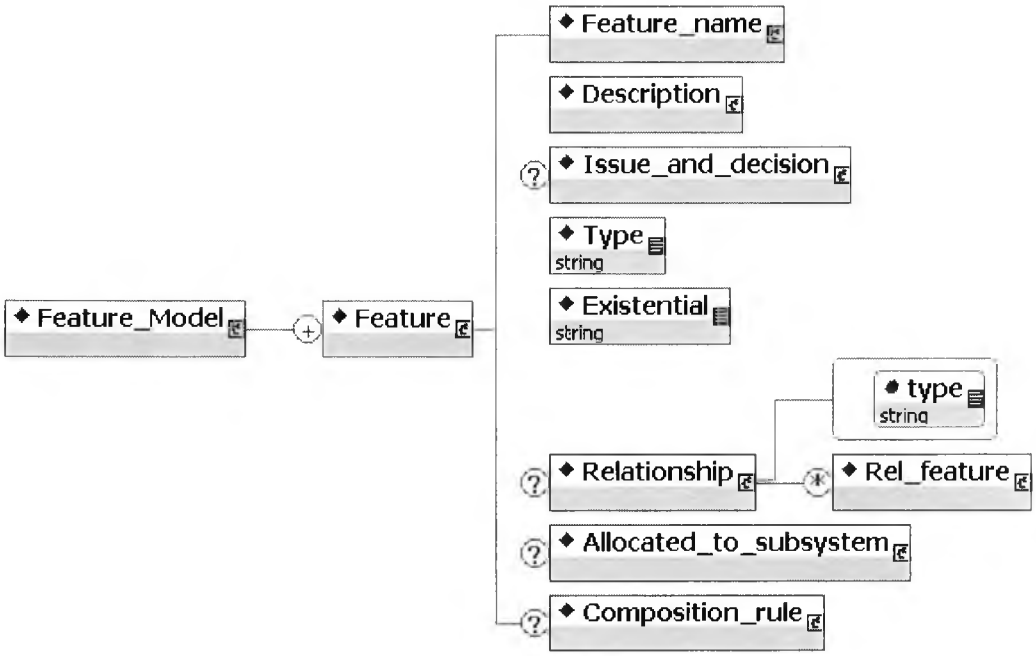
```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Rules">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="TraceRule" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Query" type="xs:string"/>
              <xs:element name="Action">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Relation">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="Element" minOccurs="2" maxOccurs="2">
                            <xs:complexType mixed="true">
                              <xs:sequence>
                                <xs:element name="Description" type="xs:string
                                  " minOccurs="0" maxOccurs="unbounded"/>
                                <xs:element name="Class" type="xs:string" minOccurs="0"/>
                                <xs:element name="Object" type="xs:string" minOccurs="0"/>
                                <xs:element name="Child" type="xs:string" minOccurs="0"/>
                                <xs:element name="Link" type="xs:string" minOccurs="0"/>
                                <xs:element name="State" type="xs:string" minOccurs="0"/>
                                <xs:element name="IncomingTransition" type="xs:string"
                                  minOccurs="0"/>
                                <xs:element name="Operation" type="xs:string" minOccurs="0"/>
                                <xs:element name="System" type="xs:string" minOccurs="0"/>
                                <xs:element name="ProcessModelID" type="xs:string"
                                  minOccurs="0"/>
                                <xs:element name="ModuleModelID" type="xs:string"
                                  minOccurs="0"/>
                                <xs:element name="Preconditions" type="xs:string"
                                  minOccurs="0"/>
                                <xs:element name="Event" type="xs:string" minOccurs="0"/>
                                <xs:element name="Transition" type="xs:string" minOccurs="0"/>
                                <xs:element name="Parameter" type="xs:string" minOccurs="0"/>
                              </xs:sequence>
                              <xs:attribute name="Document" type="xs:string" use="optional"/>
                            </xs:complexType>
                          </xs:element>
                        </xs:sequence>
                        <xs:attribute name="RuleID" type="xs:string" use="required"/>
                        <xs:attribute name="Type" type="xs:string" use="required"/>
                        <xs:attribute name="DocType1" type="xs:string" use="optional"/>
                        <xs:attribute name="DocType2" type="xs:string" use="optional"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType> </xs:element> </xs:sequence>
              <xs:attribute name="RuleID" type="xs:string" use="required"/>
              <xs:attribute name="RuleType" type="xs:string" use="required"/>
              <xs:attribute name="DocType1" type="xs:string" use="required"/>
              <xs:attribute name="DocType2" type="xs:string" use="required"/>
  </xs:complexType> </xs:element> </xs:sequence> </xs:complexType></xs:element> </xs:schema>
```

A.2. XML Schema for indirect traceability rules



```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Rules">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="TraceRule" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Query" type="xs:string"/>
              <xs:element name="Action">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Relation">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="Element" type="xs:string" minOccurs="2"
                            maxOccurs="2"/>
                          <xs:choice maxOccurs="2">
                            <xs:element name="Dependency" type="xs:string" minOccurs="0"
                              maxOccurs="2"/>
                            <xs:element name="Containment" type="xs:string" minOccurs="0"
                              maxOccurs="2"/>
                            <xs:element name="Evolution" type="xs:string" minOccurs="0"
                              maxOccurs="2"/>
                            <xs:element name="Implements" type="xs:string" minOccurs="0"
                              maxOccurs="2"/>
                            <xs:element name="Overlap" type="xs:string" minOccurs="0"
                              maxOccurs="2"/>
                            <xs:element name="Satisfiability" type="xs:string" minOccurs="0"
                              maxOccurs="2"/>
                            <xs:element name="Refinement" type="xs:string" minOccurs="0"
                              maxOccurs="2"/>
                          </xs:choice>
                          <xs:element name="VariantOf" type="xs:string" minOccurs="0"/>
                        </xs:sequence>
                        <xs:attribute name="RuleID" type="xs:string" use="required"/>
                        <xs:attribute name="Type" type="xs:string" use="required"/>
                        <xs:attribute name="Term" type="xs:string" use="required"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
            <xs:attribute name="RuleID" type="xs:string" use="required"/>
            <xs:attribute name="RuleType" type="xs:string" use="required"/>
            <xs:attribute name="DocType1" type="xs:string" use="required"/>
            <xs:attribute name="DocType2" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence></xs:complexType> </xs:element></xs:schema>
```

A.3. XML Schema for Feature model



```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XMLSPY v2004 rel. 4 U (http://www.xmlspy.com) by Waraporn (Jirapanthong) -->
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Feature_Model">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Feature" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Feature_name">
                <xs:complexType>
                  <xs:choice maxOccurs="unbounded">
                    <xs:element name="PRP" type="xs:string" minOccurs="0"/>
                    <xs:element name="JJ" type="xs:string" minOccurs="0"/>
                    <xs:element name="CC" type="xs:string" minOccurs="0"/>
                    <xs:element name="DD1" type="xs:string" minOccurs="0"/>
                    <xs:element name="VBZ" type="xs:string" minOccurs="0"/>
                    <xs:element name="TO" type="xs:string" minOccurs="0"/>
                    <xs:element name="VVI" type="xs:string" minOccurs="0"/>
                    <xs:element name="VVZ" type="xs:string" minOccurs="0"/>
                    <xs:element name="RG" type="xs:string" minOccurs="0"/>
                    <xs:element name="DB" type="xs:string" minOccurs="0"/>
                    <xs:element name="II" type="xs:string" minOccurs="0"/>
                    <xs:element name="II2" type="xs:string" minOccurs="0"/>
                    <xs:element name="AT" type="xs:string" minOccurs="0"/>
                    <xs:element name="AT0" type="xs:string" minOccurs="0"/>
                    <xs:element name="NN1" type="xs:string" minOccurs="0"/>
                    <xs:element name="NN2" type="xs:string" minOccurs="0"/>
                    <xs:element name="VM0" type="xs:string" minOccurs="0"/>
                    <xs:element name="VVI" type="xs:string" minOccurs="0"/>
                    <xs:element name="CJS" type="xs:string" minOccurs="0"/>
                    <xs:element name="CJC" type="xs:string" minOccurs="0"/>
                    <xs:element name="VBB" type="xs:string" minOccurs="0"/>
                    <xs:element name="VVB" type="xs:string" minOccurs="0"/>
                    <xs:element name="AJ0" type="xs:string" minOccurs="0"/>
                    <xs:element name="SC" type="xs:string" minOccurs="0"/>
                    <xs:element name="lO" type="xs:string" minOccurs="0"/>
                    <xs:element name="AJ0" type="xs:string" minOccurs="0"/>
                    <xs:element name="AJC" type="xs:string" minOccurs="0"/>
                    <xs:element name="AJS" type="xs:string" minOccurs="0"/>
                    <xs:element name="AT0" type="xs:string" minOccurs="0"/>
                    <xs:element name="AT1" type="xs:string" minOccurs="0"/>
                    <xs:element name="A /P" type="xs:string" minOccurs="0"/>
                    <xs:element name="AVQ" type="xs:string" minOccurs="0"/>
                    <xs:element name="CJC" type="xs:string" minOccurs="0"/>
                    <xs:element name="CJS" type="xs:string" minOccurs="0"/>
                    <xs:element name="CJT" type="xs:string" minOccurs="0"/>
                    <xs:element name="CRD" type="xs:string" minOccurs="0"/>
                    <xs:element name="DPS" type="xs:string" minOccurs="0"/>
                    <xs:element name="DT0" type="xs:string" minOccurs="0"/>
                    <xs:element name="DTQ" type="xs:string" minOccurs="0"/>
                    <xs:element name="EX0" type="xs:string" minOccurs="0"/>
                    ...
```

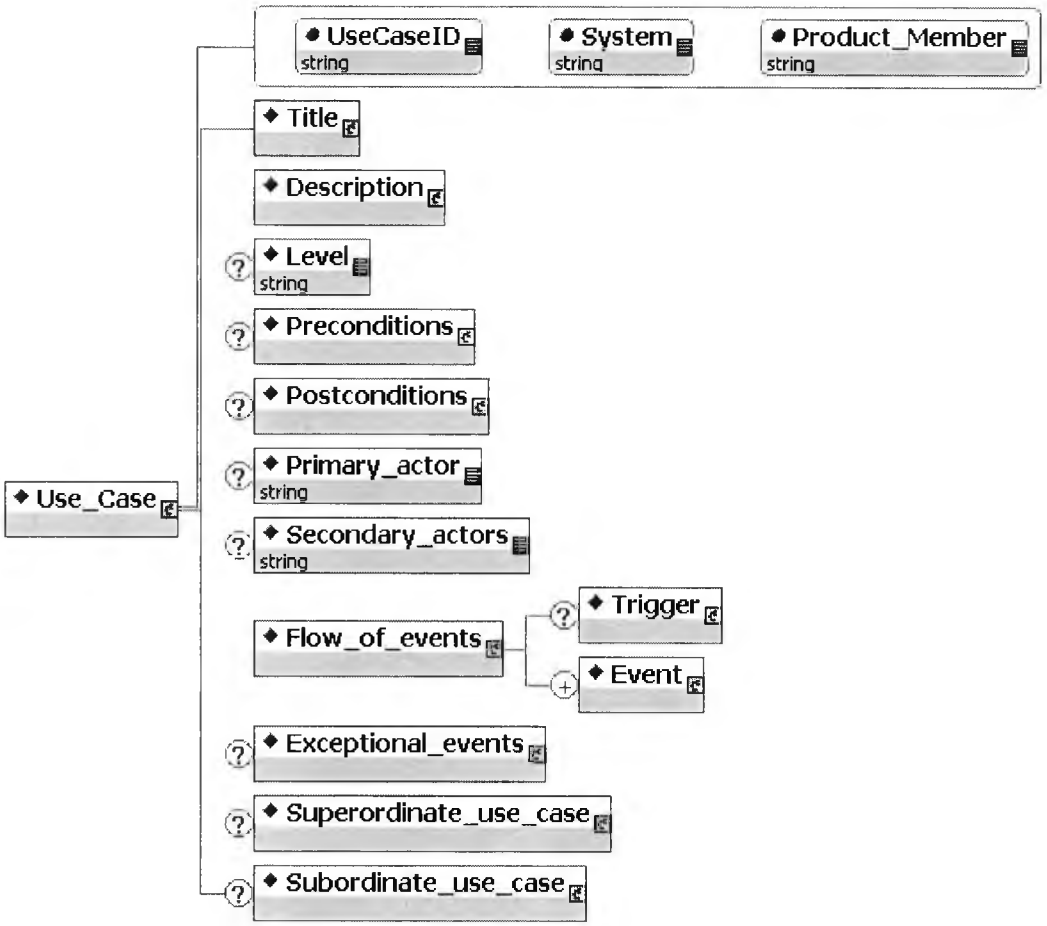
```
...
<xs:element name="ITJ" type="xs:string" minOccurs="0"/>
<xs:element name="VVB" type="xs:string" minOccurs="0"/>
<xs:element name="NN0" type="xs:string" minOccurs="0"/>
<xs:element name="NN1" type="xs:string" minOccurs="0"/>
<xs:element name="NN2" type="xs:string" minOccurs="0"/>
<xs:element name="NP0" type="xs:string" minOccurs="0"/>
<xs:element name="NULL" type="xs:string" minOccurs="0"/>
<xs:element name="ORD" type="xs:string" minOccurs="0"/>
<xs:element name="PNI" type="xs:string" minOccurs="0"/>
<xs:element name="PNP" type="xs:string" minOccurs="0"/>
<xs:element name="PNX" type="xs:string" minOccurs="0"/>
<xs:element name="PNQ" type="xs:string" minOccurs="0"/>
<xs:element name="POS" type="xs:string" minOccurs="0"/>
<xs:element name="PRF" type="xs:string" minOccurs="0"/>
<xs:element name="PUL" type="xs:string" minOccurs="0"/>
<xs:element name="PUN" type="xs:string" minOccurs="0"/>
<xs:element name="PUQ" type="xs:string" minOccurs="0"/>
<xs:element name="PUR" type="xs:string" minOccurs="0"/>
<xs:element name="TO0" type="xs:string" minOccurs="0"/>
<xs:element name="UNC" type="xs:string" minOccurs="0"/>
<xs:element name="VBB" type="xs:string" minOccurs="0"/>
<xs:element name="VBD" type="xs:string" minOccurs="0"/>
<xs:element name="VBG" type="xs:string" minOccurs="0"/>
<xs:element name="VBI" type="xs:string" minOccurs="0"/>
<xs:element name="VBZ" type="xs:string" minOccurs="0"/>
<xs:element name="VDB" type="xs:string" minOccurs="0"/>
<xs:element name="VDD" type="xs:string" minOccurs="0"/>
<xs:element name="VDG" type="xs:string" minOccurs="0"/>
<xs:element name="VDI" type="xs:string" minOccurs="0"/>
<xs:element name="VDN" type="xs:string" minOccurs="0"/>
<xs:element name="VDZ" type="xs:string" minOccurs="0"/>
<xs:element name="VHB" type="xs:string" minOccurs="0"/>
<xs:element name="VHD" type="xs:string" minOccurs="0"/>
<xs:element name="VHG" type="xs:string" minOccurs="0"/>
<xs:element name="VHI" type="xs:string" minOccurs="0"/>
<xs:element name="VHN" type="xs:string" minOccurs="0"/>
<xs:element name="VHZ" type="xs:string" minOccurs="0"/>
<xs:element name="VM0" type="xs:string" minOccurs="0"/>
<xs:element name="VV0" type="xs:string" minOccurs="0"/>
<xs:element name="VVB" type="xs:string" minOccurs="0"/>
<xs:element name="VVD" type="xs:string" minOccurs="0"/>
<xs:element name="VVG" type="xs:string" minOccurs="0"/>
<xs:element name="VVI" type="xs:string" minOccurs="0"/>
<xs:element name="VVN" type="xs:string" minOccurs="0"/>
<xs:element name="VVZ" type="xs:string" minOccurs="0"/>
<xs:element name="XX0" type="xs:string" minOccurs="0"/>
<xs:element name="ZZ0" type="xs:string" minOccurs="0"/>
<xs:element name="IF" type="xs:string" minOccurs="0"/>
<xs:element name="IW" type="xs:string" minOccurs="0"/>
<xs:element name="REX" type="xs:string" minOccurs="0"/>
<xs:element name="NP1" type="xs:string" minOccurs="0"/>
<xs:element name="PPH1" type="xs:string" minOccurs="0"/>
<xs:element name="MC" type="xs:string" minOccurs="0"/>
<xs:element name="NN" type="xs:string" minOccurs="0"/>
<xs:element name="NNU2" type="xs:string" minOccurs="0"/>
...
```

```
...
<xs:element name="VBR" type="xs:string" minOccurs="0"/>
<xs:element name="RL" type="xs:string" minOccurs="0"/>
<xs:element name="RPK" type="xs:string" minOccurs="0"/>
<xs:element name="RR" type="xs:string" minOccurs="0"/>
<xs:element name="AV0" type="xs:string" minOccurs="0"/>
<xs:element name="VBN" type="xs:string" minOccurs="0"/>
<xs:element name="CS" type="xs:string" minOccurs="0"/>
<xs:element name="VM" type="xs:string" minOccurs="0"/>
<xs:element name="CST" type="xs:string" minOccurs="0"/>
<xs:element name="XX" type="xs:string" minOccurs="0"/>
<xs:element name="APPGE" type="xs:string" minOccurs="0"/>
<xs:element name="MD" type="xs:string" minOccurs="0"/>
<xs:element name="JK" type="xs:string" minOccurs="0"/>
<xs:element name="RP" type="xs:string" minOccurs="0"/>
<xs:element name="FU" type="xs:string" minOccurs="0"/>
<xs:element name="DD" type="xs:string" minOccurs="0"/>
<xs:element name="DD1" type="xs:string" minOccurs="0"/>
<xs:element name="DD2" type="xs:string" minOccurs="0"/>
<xs:element name="NNU" type="xs:string" minOccurs="0"/>
<xs:element name="RT" type="xs:string" minOccurs="0"/>
<xs:element name="RRR" type="xs:string" minOccurs="0"/>
<xs:element name="BCL" type="xs:string" minOccurs="0"/>
<xs:element name="NNT1" type="xs:string" minOccurs="0"/>
<xs:element name="NNT2" type="xs:string" minOccurs="0"/>
<xs:element name="MC1" type="xs:string" minOccurs="0"/>
<xs:element name="MC2" type="xs:string" minOccurs="0"/>
</xs:choice>
</xs:complexType>
</xs:element>
<xs:element name="Description">
<xs:complexType>
<xs:choice maxOccurs="unbounded">
...*
</xs:choice>
</xs:complexType>
</xs:element>
<xs:element name="Issue_and_decision" minOccurs="0">
<xs:complexType>
<xs:choice maxOccurs="unbounded">
...*
</xs:choice>
</xs:complexType>
</xs:element>
<xs:element name="Type" type="xs:string"/>
<xs:element name="Existential" type="xs:string"/>
<xs:element name="Relationship" minOccurs="0">
<xs:complexType>
<xs:sequence>
<xs:element name="Rel_feature" minOccurs="0" maxOccurs="unbounded">
<xs:complexType>
<xs:choice maxOccurs="unbounded">
...*
</xs:choice>
</xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="type" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
...
```

```
                    ...
                    <xs:element name="Allocated_to_system" minOccurs="0">
                      <xs:complexType>
                        <xs:choice maxOccurs="unbounded">
                          ...*
                        </xs:choice>
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="Composition_rule" minOccurs="0">
                      <xs:complexType>
                        <xs:choice maxOccurs="unbounded">
                          ...*
                        </xs:choice>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:schema>
```

Note: <xs: choice> has sub elements e.g. PRP, JJ, CC, DD1, etc. as shown earlier.

A.4. XML Schema for use case

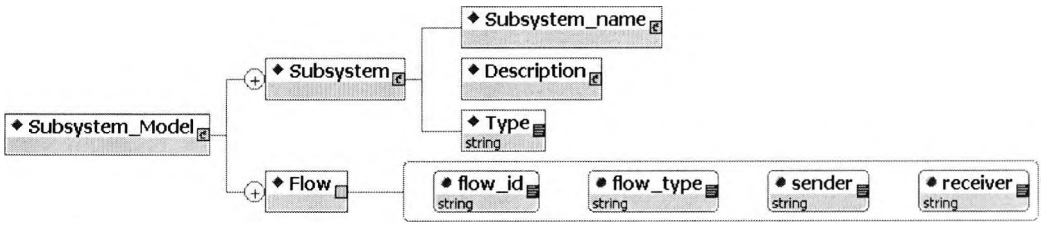


```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Use_Case">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Title">
          <xs:complexType>
            <xs:choice maxOccurs="unbounded">
              ...*
            </xs:choice>
          </xs:complexType>
        </xs:element>
        <xs:element name="Description">
          <xs:complexType>
            <xs:choice maxOccurs="unbounded">
              ...*
            </xs:choice>
          </xs:complexType>
        </xs:element>
        <xs:element name="Level" type="xs:string" minOccurs="0"/>
        <xs:element name="Preconditions" minOccurs="0">
          <xs:complexType>
            <xs:choice maxOccurs="unbounded">
              ...*
            </xs:choice>
          </xs:complexType>
        </xs:element>
        <xs:element name="Postconditions" minOccurs="0">
          <xs:complexType>
            <xs:choice maxOccurs="unbounded">
              ...*
            </xs:choice>
          </xs:complexType>
        </xs:element>
        <xs:element name="Primary_actor">
          <xs:complexType>
            <xs:choice maxOccurs="unbounded">
              ...*
            </xs:choice>
          </xs:complexType>
        </xs:element>
        <xs:element name="Secondary_actors" minOccurs="0">
          <xs:complexType>
            <xs:choice maxOccurs="unbounded">
              ...*
            </xs:choice>
          </xs:complexType>
        </xs:element>
        ...
```

```
...
<xs:element name="Flow_of_events">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Trigger" minOccurs="0">
        <xs:complexType>
          <xs:choice maxOccurs="unbounded">
            ...*
          </xs:choice>
        </xs:complexType>
      </xs:element>
      <xs:element name="Event" maxOccurs="unbounded">
        <xs:complexType>
          <xs:choice maxOccurs="unbounded">
            ...*
          </xs:choice>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="Exceptional_events" minOccurs="0">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      ...*
    </xs:choice>
  </xs:complexType>
</xs:element>
<xs:element name="Superordinate_use_case" minOccurs="0">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      ...*
    </xs:choice>
  </xs:complexType>
</xs:element>
<xs:element name="Subordinate_use_case" minOccurs="0">
  <xs:complexType>
    <xs:choice maxOccurs="unbounded">
      ...*
    </xs:choice>
  </xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="UseCaseID" type="xs:string" use="required"/>
<xs:attribute name="System" type="xs:string" use="required"/>
<xs:attribute name="Family_Member" type="xs:string" use="required"/>
</xs:complexType>
</xs:element>
</xs:schema>
```

Note: <xs: choice> has sub elements e.g. PRP, JJ, CC, DD1, etc. as shown earlier.

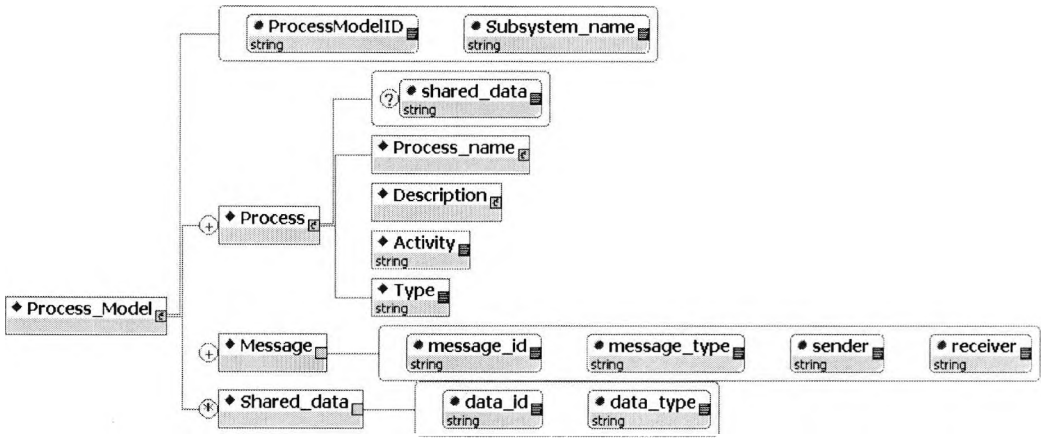
A.5. XML Schema for subsystem model




```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Subsystem_Model">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Subsystem" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Subsystem_name">
                <xs:complexType>
                  <xs:choice maxOccurs="unbounded">
                    ...*
                  </xs:choice>
                </xs:complexType>
              </xs:element>
              <xs:element name="Description">
                <xs:complexType>
                  <xs:choice maxOccurs="unbounded">
                    ...*
                  </xs:choice>
                </xs:complexType>
              </xs:element>
              <xs:element name="Type" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Flow" maxOccurs="unbounded">
          <xs:complexType>
            <xs:attribute name="flow_id" type="xs:string" use="required"/>
            <xs:attribute name="flow_type" type="xs:string" use="required"/>
            <xs:attribute name="sender" type="xs:string" use="required"/>
            <xs:attribute name="receiver" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Note: <xs: choice> has sub elements e.g. PRP, JJ, CC, DD1, etc. as shown earlier.

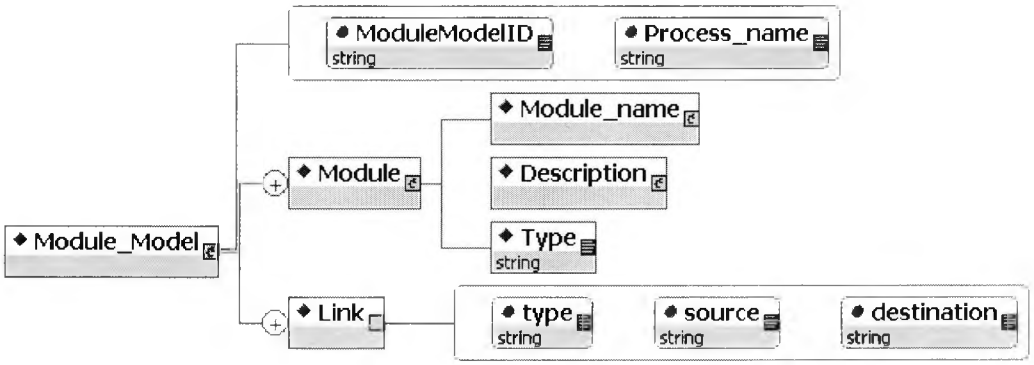
A.6. XML Schema for process model



```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Process_Model">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Process" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Process_name">
                <xs:complexType>
                  <xs:choice maxOccurs="unbounded">
                    ...*
                  </xs:choice>
                </xs:complexType>
              </xs:element>
              <xs:element name="Description">
                <xs:complexType>
                  <xs:choice maxOccurs="unbounded">
                    ...*
                  </xs:choice>
                </xs:complexType>
              </xs:element>
              <xs:element name="Activity" type="xs:string"/>
              <xs:element name="Type" type="xs:string"/>
            </xs:sequence>
            <xs:attribute name="shared_data" type="xs:string" use="optional"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="Message" maxOccurs="unbounded">
          <xs:complexType>
            <xs:attribute name="message_id" type="xs:string" use="required"/>
            <xs:attribute name="message_type" type="xs:string" use="required"/>
            <xs:attribute name="sender" type="xs:string" use="required"/>
            <xs:attribute name="receiver" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
        <xs:element name="Shared_data" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:attribute name="data_id" type="xs:string" use="required"/>
            <xs:attribute name="data_type" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="ProcessModelID" type="xs:string" use="required"/>
      <xs:attribute name="Subsystem_name" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Note: <xs: choice> has sub elements e.g. PRP, JJ, CC, DD1, etc. as shown earlier.

A.7. XML Schema for module model



```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified" attributeFormDefault="unqualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Module_Model">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Module" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Module_name">
                <xs:complexType>
                  <xs:choice maxOccurs="unbounded">
                    ...*
                  </xs:choice>
                </xs:complexType>
              </xs:element>
              <xs:element name="Description">
                <xs:complexType>
                  <xs:choice maxOccurs="unbounded">
                    ...*
                  </xs:choice>
                </xs:complexType>
              </xs:element>
              <xs:element name="Type" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="Link" maxOccurs="unbounded">
          <xs:complexType>
            <xs:attribute name="type" type="xs:string" use="required"/>
            <xs:attribute name="source" type="xs:string" use="required"/>
            <xs:attribute name="destination" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="ModuleModelID" type="xs:string" use="required"/>
      <xs:attribute name="Process_name" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Note: <xs: choice> has sub elements e.g. PRP, JJ, CC, DD1, etc. as shown earlier.

Appendix B – Traceability Rules

B.1.Direct Traceability Rules

```
<TraceRule RuleID="R10"  RuleType="satisfiability" DocType1="Subsystem Model"
DocType2="Feature Model">
   <Query>

declare namespace s="java:distanceControl.d";

declare function local:getParentFeature($child as xs:string) as item()
{
for $itemA in doc(Â£2Â£)//Relationship/Rel_feature
where normalize-space($itemA)= normalize-space($child)
return $itemA/../../Feature_name
};

for $item1 in doc(Â£1Â£)//Subsystem/Description,
 $item2 in doc(Â£2Â£)//Feature_Model/Feature/Feature_name
where local:getParentFeature(string($item2)) != ""
and s:containsInDistance($item1,$item2, local:getParentFeature(string($item2)))

   </Query>
   <Action>
      <Relation RuleID="R10" Type="satisfiability" DocType1="Subsystem Model"
                DocType2="Feature Model">
         <Element Document=""> {$item1/../Subsystem_name} </Element>
         <Element Document=""> {$item2} </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R11"  RuleType="implements" DocType1="Class Diagram"
DocType2="Feature Model">
   <Query>

declare namespace UML="org.omg.xmi.namespace.UML";
declare namespace s="java:distanceControl.d";

declare function local:getClassinClass($diagram as  xs:string) as item()*
{
for $itemE in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
where
$itemE/../../../../../../@name = $diagram
return $itemE
};

let $cl := local:getClassinClass(*1*)
```

```
for $item0 in $cl

for $item1 in
doc(Â£1Â£)//UML:Namespace.ownedElement/UML:Class/UML:Classifier.feature/UML:Operat
ion
for $item2 in doc(Â£2Â£)//Feature_Model/Feature/Description
let $t1 := $item1/../../@name

where   $item1/../../@xmi.id = $item0/@xmi.idref
and s:containsInDistance($item2, $item1/@name, $t1)

   </Query>
   <Action>
      <Relation RuleID="R11" Type="implements" DocType1="Class Diagram"
DocType2="Feature Model">
         <Element Document=""><Class>{$t1}</Class>
                                  <Operation>{$item1/@name}</Operation></Element>
         <Element Document=""> {$item2/../Feature_name} </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R12"  RuleType="dependency" DocType1="Use Case"
DocType2="Feature Model">
   <Query>

declare namespace s="java:distanceControl.d";

declare function local:getChildrenFeature($parent as xs:string) as item()*
{
for $itemA in doc(Â£2Â£)//Feature/Relationship/Rel_feature
where normalize-space($itemA/../../Feature_name)= normalize-space(Sparent)
return $itemA
};

for $item1 in doc(Â£1Â£)//Use_Case/Preconditions,
 $item2 in doc(Â£2Â£")//Feature_Model/Feature/Feature_name

where local:getChildrenFeature(string($item2)) != ""
and s:containsInDistance($item1,$item2, local:getChildrenFeature(string($item2)))

   </Query>
   <Action>
      <Relation RuleID="R12" Type="dependency" DocType1="Use Case"
                DocType2="Feature Model">
         <Element Document=""> {$item1/../Title} <Preconditions/>
         </Element>
         <Element Document=""> {$item2}
              <Child> { local:getChildrenFeature(string($item2))}</Child>
         </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R13"  RuleType="dependency" DocType1="Process Model"
DocType2="Feature Model">
```

```
  <Query>

declare namespace w="java:synonym.s";

declare function local:getFeatureofSubsystem($subsystem as xs:string) as item()*
{
for $itemA in doc(Â£2Â£)//Feature_Model/Feature/Allocated_to_Subsystem
where normalize-space($itemA)= normalize-space($subsystem)
return $itemA/SitemA/../Feature_name
};

for $item1 in doc(Â£1Â£)//Process_Model
let $t2 := local:getFeatureofSubsystem(string($item1/@Subsystem_name))

  </Query>
  <Action>
    <Relation RuleID="R13" Type="dependency" DocType1="Process Model"
              DocType2="Feature Model">
      <Element Document="">
          <ProcessModelID>{$item1/@ProcessModelID}</ProcessModelID> </Element>
      <Element Document=""> {$t2} </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R14"  RuleType="containment" DocType1="Use Case"
DocType2="Feature Model">
  <Query>

declare namespace s="java:distanceControl.d";

declare function local:getParentFeature($child as xs:string) as item()
{
for $itemA in doc(Â£2Â£)//Relationship/Rel_feature
where normalize-space($itemA)= normalize-space($child)
return $itemA/../../Feature_name
};

for $item1 in doc(Â£1Â£)//Use_Case/Description,
 $item2 in doc(Â£2Â£)//Feature_Model/Feature/Feature_name
where s:containsInDistance($item1,$item2, local:getParentFeature(string($item2)))

 </Query>
  <Action>
    <Relation RuleID="R14" Type="containment" DocType1="Use Case"
              DocType2="Feature Model">
      <Element Document=""> {$item1/../Title} <Description/>
      </Element>
      <Element Document=""> {$item2} </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R15"  RuleType="containment" DocType1="Process Model"
           DocType2="Class Diagram">
  <Query>
declare namespace UML="org.omg.xmi.namespace.UML";
```

```
declare namespace s="java:distanceControl.d";

for $item1 in doc(Â£2Â£)//UML:Classifier.feature/UML:Operation/@name
for $item2 in doc(Â£1Â£)//Process/Description
let $t1 := $item1/../../../@name

where s:containsInDistance($item2,$item1, $t1)
</Query>
  <Action>
    <Relation RuleID="R15" Type="containment" DocType1="Process Model"
              DocType2="Class Diagram">
      <Element Document=""> {$item2/../Process_name} <Description/>
      </Element>
      <Element Document=""> <Class>{$t1}</Class> <Operation> {$item1} </Operation>
      </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R16"  RuleType="refinement" DocType1="Module Model"
DocType2="Process Model">
  <Query>

declare namespace w="java:synonym.s";

for $item1 in doc(Â£2Â£)//Process_Model/Process
let $item2 := doc(Â£1Â£)//Module_Model
where w:stringnospace(string($item1/Process_name))=
w:stringnospace($item2/@Process_name)

</Query>
  <Action>
    <Relation RuleID="R16" Type="refinement" DocType1="Module Model"
              DocType2="Process Model">
      <Element Document=""> <ModuleModelID>{$item2/@ModuleModelID}
                                  </ModuleModelID></Element>
      <Element Document=""> {$item1/Process_name} </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R17"  RuleType="overlaps" DocType1="Statechart Diagram"
DocType2="Use Case">
  <Query>

declare namespace UML="org.omg.xmi.namespace.UML";
declare namespace s="java:distanceControl.d";

declare function local:getTransitioninState() as item()*
{
for $itemF in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Transition
return $itemF
};

declare function local:getStateinState($transition as node()) as item()
```

```
{
for $itemG in doc(Â£1Â£)//UML:SimpleState
where $itemG/UML:StateVertex.incoming/UML:Transition/@xmi.idref =
$transition/@xmi.idref
return $itemG
};

let $item1 := local:getTransitioninState()
for $t1 in $item1
for $item2 in local:getStateinState($t1)
for $item3 in doc(Â£2Â£)//Use_Case/Flow_of_events/Event
where s:containsInDistance($item3, $item2/@name)

</Query>
  <Action>
    <Relation RuleID="R17" Type="overlaps" DocType1="Statechart Diagram"
              DocType2="Use Case">
      <Element Document=""> <State>{$item2/@name} </State>
            <IncomingTransition>{$t1/@xmi.idref}</IncomingTransition> </Element>
      <Element Document=""> {$item3/../../Title} <Event/>
      </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R9"  RuleType="evolution" DocType1="Statechart Diagram"
DocType2="Statechart Diagram">
  <Query>
for $item1 in doc(Â£1Â£)//UML:Transition,
 $item2 in doc(Â£2Â£)//UML:Transition

where
$x1/../UML:Diagram/@name = $x2/../ UML:Diagram/@name  and  $x1/@name = $x2/@name
and
$x1/UML:Transition.effect/UML:ActionSequence/UML:ActionSequence.action/UML:Uninterpr
etedAction/@name =
$x2/UML:Transition.effect/UML:ActionSequence/UML:ActionSequence.action/UML:Uninterpr
etedAction/@name and
$x1/UML:Transition.trigger/Behavioral_Elements.State_Machines.Event/@xmi.idref =
$x1/../UML:SignalEvent/@xmi.id and
$x2/UML:Transition.trigger/Behavioral_Elements.State_Machines.Event/@xmi.idref =
$x2/../UML:SignalEvent/@xmi.id and
$x1/../UML:SignalEvent/@name = $x2/../UML:SignalEvent/@name and
$x1/../UML:SignalEvent/UML:SignalEvent.signal/Behavioral_Elements.Common_Behavior.Sig
nal/@xmi.idref = $x1/../UML:Signal/@xmi.id and
$x2/../UML:SignalEvent/UML:SignalEvent.signal/Behavioral_Elements.Common_Behavior.Sig
nal/@xmi.idref = $x2/../UML:Signal/@xmi.id and
$x1/../UML:Signal/@name = $x2/../UML:Signal/@name and
$x1/../UML:Signal/@xmi.id/UML:DataType/@xmi.id =
$x1/../UML:Event.Parameter/UML:Parameter/UML:Parameter.type/Foundation.Core.Classifier/
@xmi.idref and  $x2/../UML:Signal/@xmi.id/UML:DataType/@xmi.id =
$x2/../UML:Event.Parameter/UML:Parameter/UML:Parameter.type/Foundation.Core.Classifier/
@xmi.idrefand  $x1/../UML:Event.Parameter/UML:Parameter/@name !=
$x2/../UML:Event.Parameter/UML:Parameter/@name

</Query>
```

```
  <Action>
    <Relation RuleID="R9" Type="evolution" DocType1="Statechart Diagram"
              DocType2="Statechart Diagram">
      <Element Document="">
        <Transition>{$x1/@name} </Transition>
        <Parameter> {$x1/../UML:Event.Parameter/UML:Parameter/@name } </Parameter>
      </Element>
      <Element Document="">
        <Transition> {$x2/@name } </Transition>
        <Parameter> {$x2/../UML:Event.Parameter/UML:Parameter/@name} </Parameter>
      </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R18"  RuleType="implements" DocType1="Sequence Diagram"
DocType2="Use Case">
  <Query>
declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

declare function local:getMessageinSeq() as item()*
{
for $itemA in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Link
return $itemA

};

declare function local:getObjectinSeq($link as node()) as item()
{
for $itemB in doc(Â£1Â£)//UML:Link
where $itemB/@xmi.id = $link/@xmi.idref
return $itemB/UML:Link.connection/UML:LinkEnd/UML:LinkEnd.instance/UML:Object
};

declare function local:getObjectinModel($object as node()) as item()*
{

for $itemC in doc(Â£1Â£)//UML:Object
where $itemC/@xmi.id = $object/@xmi.idref
return $itemC/UML:Instance.classifier/UML:Class
};

declare function local:getClassObjectinSeq($class as node()) as item()
{
for $itemD in doc(Â£1Â£)//UML:Class
where $itemD/@xmi.id = $class/@xmi.idref

return $itemD

};

declare function local:getClassinClass($class as node(), $diagram as xs:string)as item()*
```

```
{
for $itemE in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
where $itemE/../../../../../parent::node()/@name = $diagram
return $itemE
};

let $item1 := local:getMessageinSeq()
for $t1 in $item1
for $item2 in local:getObjectinSeq($t1)
for $item3 in local:getObjectinModel($item2)
for $item4 in local:getClassObjectinSeq($item3)

for $item6 in doc(Â£2Â£)//Use_Case/Description

where s:containsInDistance($item6, $item4/@name)

</Query>
  <Action>
    <Relation RuleID="R18" Type="implements" DocType1="Sequence Diagram"
              DocType2="Use Case"   >
      <Element Document=""> <Class>{$item4/@name}</Class> <Link>{$t1}</Link>
                                                          </Element>
      <Element Document="">{$item6/../Title} <Description/></Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R19"  RuleType="refinement" DocType1="Sequence Diagram"
DocType2="Use Case">
  <Query>
declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

declare function local:getMessageinSeq() as item()*
{
for $itemA in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Link
return $itemA

};

declare function local:getObjectinSeq($link as node()) as item()
{
for $itemB in doc(Â£1Â£)//UML:Link
where $itemB/@xmi.id = $link/@xmi.idref
return $itemB/UML:Link.connection/UML:LinkEnd/UML:LinkEnd.instance/UML:Object
};

declare function local:getObjectinModel($object as node()) as item()*
{
```

```
for $itemC in doc(Â£1Â£)//UML:Object
where $itemC/@xmi.id = $object/@xmi.idref
return $itemC/UML:Instance.classifier/UML:Class
};

declare function local:getClassObjectinSeq($class as node()) as item()
{
for $itemD in doc(Â£1Â£)//UML:Class
where $itemD/@xmi.id = $class/@xmi.idref

return $itemD

};

declare function local:getClassinClass($class as node(), $diagram as xs:string)as item()*
{
for $itemE in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
where $itemE/../../../../../parent::node()/@name = $diagram
return $itemE
};

let $item1 := local:getMessageinSeq()
for $t1 in $item1
for $item2 in local:getObjectinSeq($t1)
for $item3 in local:getObjectinModel($item2)
for $item4 in local:getClassObjectinSeq($item3)

for $item6 in doc(Â£2Â£)//Use_Case/Description

where s:containsInDistance($item6, $item4/@name)

</Query>
  <Action>
    <Relation RuleID="R19" Type="refinement" DocType1="Sequence Diagram"
              DocType2="Use Case">
      <Element Document=""><Class>{$item4/@name}</Class><Link>{$t1}</Link>
                                                          </Element>
      <Element Document="">{$item6/../Title}<Description/> </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R20"  RuleType="satisfiability" DocType1="Sequence Diagram"
DocType2="Use Case">
  <Query>
declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

declare function local:getMessageinSeq() as item()*
{
```

```
for $itemA in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Link
return $itemA

};

declare function local:getObjectinSeq($link as node()) as item()
{
for $itemB in doc(Â£1Â£)//UML:Link
where $itemB/@xmi.id = $link/@xmi.idref
return $itemB/UML:Link.connection/UML:LinkEnd/UML:LinkEnd.instance/UML:Object
};

declare function local:getObjectinModel($object as node()) as item()*
{

for $itemC in doc(Â£1Â£)//UML:Object
where $itemC/@xmi.id = $object/@xmi.idref
return $itemC/UML:Instance.classifier/UML:Class
};

declare function local:getClassObjectinSeq($class as node()) as item()
{
for $itemD in doc(Â£1Â£)//UML:Class
where $itemD/@xmi.id = $class/@xmi.idref

return $itemD

};

declare function local:getClassinClass($class as node(), $diagram as xs:string)as item()*
{
for $itemE in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
where $itemE/../../../../../parent::node()/@name = $diagram
return $itemE
};

let $item1 := local:getMessageinSeq()
for $t1 in $item1
for $item2 in local:getObjectinSeq($t1)
for $item3 in local:getObjectinModel($item2)
for $item4 in local:getClassObjectinSeq($item3)

for $item6 in doc(Â£2Â£)//Use_Case/Description

where s:containsInDistance($item6, $item4/@name)

  </Query>
  <Action>
    <Relation RuleID="R20" Type="satisfiability" DocType1="Sequence Diagram"
              DocType2="Use Case">
```

```
        <Element Document=""> <Class>{$item4/@name}</Class> <Link>{$t1}</Link>
        </Element>
        <Element Document="">{$item6/../Title}<Description/>
        </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R21"  RuleType="implements" DocType1="Class Diagram"
DocType2="Use Case">
   <Query>
declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

for $itemA in doc(Â£1Â£)//UML:Class

for $item1 in doc(Â£1Â£)//UML:Classifier.feature/UML:Operation/@name
for $item2 in doc(Â£2Â£)//Use_Case

let $t1 := $item1/../../../@name

where
s:containsInDistance($item2/Title, $t1)
and
s:containsInDistance($item2/Description,$item1)

   </Query>
   <Action>
      <Relation RuleID="R21" Type="implements" DocType1="Class Diagram"
                DocType2="Use Case">
        <Element Document=""> <Class>{$t1}</Class><Operation>{$item1}</Operation>
        </Element>
        <Element Document="">{$item2/Title}<Description/>
        </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R22"  RuleType="containment" DocType1="Use Case"
DocType2="Feature Model">
   <Query>

declare namespace s="java:distanceControl.d";

for
 $item1 in doc(Â£1Â£)//Use_Case,
 $item2 in doc(Â£2Â£)//Feature_Model/Feature

where
s:containsInDistance($item1/Description, $item2/Feature_name)

</Query>
   <Action>
      <Relation RuleID="R22" Type="containment" DocType1="Use Case"
                DocType2="Feature Model">
        <Element Document=""> {$item1/Title} </Element>
```

```
        <Element Document=""> {$item2/Feature_name} </Element>
      </Relation>
    </Action>
</TraceRule>
<TraceRule RuleID="R23"  RuleType="refinement" DocType1="Statechart Diagram"
DocType2="Sequence Diagram">
    <Query>

declare namespace UML="org.omg.xmi.namespace.UML";

declare function local:getOperationinSeq() as item()*
{
for $itemA in
doc(Â£2Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Operation
return $itemA

};

declare function local:getOperationinModel($operation as node()) as item()*
{

for $itemC in doc(Â£2Â£)//UML:Classifier.feature/UML:Operation
where $itemC/@xmi.id = $operation/@xmi.idref

return $itemC

};

declare function local:getStateofOperationinState($operation as node()) as item()
{
for $itemD in doc(Â£1Â£)//UML:SimpleState
where $itemD/@name = $operation/@name
return $itemD

};


for $item1 in local:getOperationinSeq()
for $item2 in local:getOperationinModel($item1)
for $item3 in local:getStateofOperationinState($item2)

</Query>
    <Action>
      <Relation RuleID="R23" Type="refinement" DocType1="Statechart Diagram"
                DocType2="Sequence Diagram">
        <Element Document=""> <State>{$item3/@name}{$item3/@xmi.id}</State>
        </Element>
        <Element Document=""> <Object>{$item2/../../@name}</Object>
                              <Operation>{$item2/@name}</Operation> </Element>
      </Relation>
    </Action>
</TraceRule>
<TraceRule RuleID="R24"  RuleType="implement" DocType1="Class Diagram"
DocType2="Use Case">
```

```
  <Query>
      declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

for $item1 in doc(Â£1Â£)//UML:Class/@name

for $item2 in doc(Â£2Â£)//Use_Case/Description


where
s:containsInDistance($item2,$item1)


  </Query>
  <Action>
    <Relation  RuleID="R24" Type="implement" DocType1="Class Diagram"
               DocType2="Use Case">
      <Element Document=""> <Class>{$item1}</Class> </Element>
      <Element Document="">{$item2/../Title}</Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R25"  RuleType="containment" DocType1="Statechart Diagram"
DocType2="Class Diagram">
  <Query>

declare namespace s="java:distanceControl.d";
declare namespace UML="org.omg.xmi.namespace.UML";

for $itemA in doc(Â£1Â£)//UML:CompositeState.subvertex/UML:SimpleState
for $itemC in doc(Â£2Â£)//UML:Classifier.feature/UML:Operation

where s:containsInDistance($itemC/@name, $itemA/@name)

  </Query>
  <Action>
    <Relation RuleID="R25" Type="containment" DocType1="Statechart Diagram"
              DocType2="Class Diagram">
      <Element Document=""> <State>{$itemA/@name} </State></Element>
      <Element Document=""> <Class>{$itemC/../../@name}</Class>
                            <Operation>{$itemC/@name}</Operation></Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R26"  RuleType="overlap" DocType1="Class Diagram"
DocType2="Statechart Diagram">
  <Query>

declare namespace w="java:synonym.s";
declare namespace UML ="org.omg.xmi.namespace.UML";

for $item1 in doc(Â£1Â£)//UML:Classifier.feature/UML:Operation,
$item2 in doc(Â£2Â£)//UML:CompositeState.subvertex/UML:SimpleState
```

```
where
$item1/@name = $item2/@name


</Query>
  <Action>
    <Relation RuleID="R26" Type="overlap" DocType1="Class Diagram"
              DocType2="Statechart Diagram">
      <Element Document=""> <Class>{$item1/../../@name}</Class>
            <Operation>{$item1/@name} </Operation> </Element>
      <Element Document=""><State>{ $item2/@name}  </State>
      </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R27"  RuleType="overlap" DocType1="Sequence Diagram"
DocType2="Class Diagram">
  <Query>

declare namespace w="java:synonym.s";
declare namespace UML ="org.omg.xmi.namespace.UML";

for $item1 in doc(Â£1Â£)//UML:Object,
$item2 in doc(Â£2Â£)//UML:Class

where
$item1/@name = $item2/@name


</Query>
  <Action>
    <Relation RuleID="R27" Type="overlap" DocType1="Sequence Diagram"
              DocType2="Class Diagram">
      <Element Document=""> <Object>{$item1/@xmi.id} {$item1/@name} </Object>
      </Element>
      <Element Document=""> <Class>{$item2/@xmi.id}{ $item2/@name}  </Class>
      </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R28"  RuleType="refinement" DocType1="Module Model"
DocType2="Process Model">
  <Query>
  declare namespace w="java:synonym.s";

  for $item1 in doc(Â£2Â£)//Process_Model/Process
  let $item2 := doc(Â£1Â£)//Module_Model

  where
  w:stringnospace(string($item1/Process_name))= w:stringnospace($item2/@Process_name)
  </Query>
  <Action>
    <Relation RuleID="R28" Type="refinement" DocType1="Module Model"
              DocType2="Process Model">
      <Element Document=""> <ModuleModelID>{ $item2/@ModuleModelID}
```

```
                    {$item2/@Process_name}</ModuleModelID> </Element>
      <Element Document=""> { $item1/Process_name } </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R29"  RuleType="dependency" DocType1="Statechart Diagram"
DocType2="Feature Model">
  <Query>

declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

declare function local:getTransitioninState() as item()*
{
for $itemF in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Transition
return $itemF

};

declare function local:getTransitioninState($transition as node()) as item()*
{
for $itemG in doc(Â£1Â£)//UML:SimpleState
where $itemG/UML:StateVertex.incoming/UML:Transition/@xmi.idref =
$transition/@xmi.idref
return $itemG

};

let $item1 := local:getTransitioninState()
for $t1 in $item1
for $item2 in local:getTransitioninState($t1)
for $item3 in doc(Â£2Â£)//Feature/Description
where
s:containsInDistance($item3, $item2/@name)

  </Query>
  <Action>
    <Relation RuleID="R29" Type="dependency" DocType1="Statechart Diagram"
              DocType2="Feature Model">
      <Element Document=""> <State>{$item2/@name} </State>
        <IncomingTransition>{$t1/@xmi.idref}</IncomingTransition> </Element>
      <Element Document=""> {$item3/../Feature_name} </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R30"  RuleType="dependency" DocType1="Sequence Diagram"
DocType2="Feature Model">
  <Query>
declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";
```

```
declare function local:getMessageinSeq() as item()*
{
for $itemA in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Link
return $itemA

};

declare function local:getObjectinSeq($link as node()) as item()
{
for $itemB in doc(Â£1Â£)//UML:Link
where $itemB/@xmi.id = $link/@xmi.idref
return $itemB/UML:Link.connection/UML:LinkEnd/UML:LinkEnd.instance/UML:Object
};

declare function local:getObjectinModel($object as node()) as item()*
{

for $itemC in doc(Â£1Â£)//UML:Object
where $itemC/@xmi.id = $object/@xmi.idref
return $itemC/UML:Instance.classifier/UML:Class
};

declare function local:getClassObjectinSeq($class as node()) as item()
{
for $itemD in doc(Â£1Â£)//UML:Class
where $itemD/@xmi.id = $class/@xmi.idref

return $itemD

};

declare function local:getClassinClass($class as node(), $diagram as xs:string)as item()*
{
for $itemE in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
where $itemE/../../../../../parent::node()/@name = $diagram
return $itemE
};

let $item1 := local:getMessageinSeq()
for $t1 in $item1
for $item2 in local:getObjectinSeq($t1)
for $item3 in local:getObjectinModel($item2)
for $item4 in local:getClassObjectinSeq($item3)

for $item6 in doc(Â£2Â£)//Feature/Description

where s:containsInDistance($item6, $item4/@name)

  </Query>
  <Action>
```

```
    <Relation RuleID="R30" Type="dependency" DocType1="Sequence Diagram"
              DocType2="Feature Model">
      <Element Document=""> <Class>{$item4/@name}</Class><Link>{$t1}</Link>
      </Element>
      <Element Document="">{$item6/../Feature_name}<Description/>
      </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R31"  RuleType="containment" DocType1="Subsystem Model"
DocType2="Class Diagram">
  <Query>
declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

declare function local:getClassinClass($diagram as  xs:string) as item()*
{
for $itemE in
doc(Â£2Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
where
$itemE/../../../../../../@name = $diagram
return $itemE
};

let $cl := local:getClassinClass("CD_phone")

for $item0 in $cl

for $item1 in
doc(Â£2Â£)//UML:Namespace.ownedElement/UML:Class/UML:Classifier.feature/UML:Operat
ion
for $item2 in doc(Â£1Â£)//Subsystem/Description
let $t1 := $item1/../../@name

where
$item1/../../@xmi.id = $item0/@xmi.idref
and
s:containsInDistance($item2, $item1/@name)

</Query>
  <Action>
    <Relation RuleID="R31" Type="containment" DocType1="Subsystem Model"
              DocType2="Class Diagram">
      <Element Document="">{$item2/../Subsystem_name}</Element>
      <Element Document="">  <Class>{$t1}</Class><Operation>{$item1} </Operation>
      </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R32"  RuleType="containment" DocType1="Process Model"
DocType2="Sequence Diagram">
  <Query>
```

```
declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

declare function local:getMessageinSeq() as item()*
{
for $itemA in
doc(Â£2Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Link
return $itemA

};

declare function local:getObjectinSeq($link as node()) as item()
{
for $itemB in doc(Â£2Â£)//UML:Link
where $itemB/@xmi.id = $link/@xmi.idref
return $itemB/UML:Link.connection/UML:LinkEnd/UML:LinkEnd.instance/UML:Object
};

declare function local:getObjectinModel($object as node()) as item()*
{

for $itemC in doc(Â£2Â£)//UML:Object
where $itemC/@xmi.id = $object/@xmi.idref
return $itemC/UML:Instance.classifier/UML:Class
};

declare function local:getClassObjectinSeq($class as node()) as item()
{
for $itemD in doc(Â£2Â£)//UML:Class
where $itemD/@xmi.id = $class/@xmi.idref

return $itemD

};

declare function local:getClassinClass($class as node(), $diagram as xs:string)as item()*
{
for $itemE in
doc(Â£2Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
where $itemE/../../../../../parent::node()/@name = $diagram
return $itemE
};

let $item1 := local:getMessageinSeq()
for $t1 in $item1
for $item2 in local:getObjectinSeq($t1)
for $item3 in local:getObjectinModel($item2)
for $item4 in local:getClassObjectinSeq($item3)

for $item6 in doc(Â£1Â£)//Process_Model/Process/Description
```

```
where s:containsInDistance($item6, $item4/@name)


  </Query>
  <Action>
    <Relation RuleID="R32" Type="containment" DocType1="Process Model"
              DocType2="Sequence Diagram">
      <Element Document="">{$item6/../Process_name}<Description/>
      </Element>
      <Element Document=""><Class>{$item4/@name}</Class><Link>{$t1}</Link>
      </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R33"  RuleType="containment" DocType1="Process Model"
DocType2="Statechart Diagram">
  <Query>

declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

declare function local:getTransitioninState() as item()*
{
for $itemF in
doc(Â£2Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Transition
return $itemF

};

declare function local:getStateinState($transition as node()) as item()
{
for $itemG in doc(Â£2Â£)//UML:SimpleState
where $itemG/UML:StateVertex.incoming/UML:Transition/@xmi.idref =
$transition/@xmi.idref
return $itemG

};

let $item1 := local:getTransitioninState()
for $t1 in $item1
for $item2 in local:getStateinState($t1)
for $item3 in doc(Â£1Â£)//Process/Description
where
s:containsInDistance($item3, $item2/@name)

  </Query>
  <Action>
    <Relation RuleID="R33" Type="containment" DocType1="Process Model"
              DocType2="Statechart Diagram">
      <Element Document=""> {$item3/../Process_name} </Element>
      <Element Document=""> <State>{$item2/@name} </State>
          <IncomingTransition>{$t1/@xmi.idref}</IncomingTransition>
```

```
        </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R34"  RuleType="satisfiability" DocType1="Statechart Diagram"
DocType2="Use Case">
   <Query>
   declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

declare function local:getTransitioninState() as item()*
{
for $itemF in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Transition
return $itemF

};

declare function local:getStateinState($transition as node()) as item()
{
for $itemG in doc(Â£1Â£)//UML:SimpleState
where $itemG/UML:StateVertex.incoming/UML:Transition/@xmi.idref =
$transition/@xmi.idref
return $itemG

};

let $item1 := local:getTransitioninState()
for $t1 in $item1
for $item2 in local:getStateinState($t1)
for $item3 in doc(Â£2Â£)//Use_Case/Description
where
s:containsInDistance($item3, $item2/@name)
   </Query>
   <Action>
      <Relation RuleID="R34" Type="satisfiability" DocType1="Statechart Diagram"
               DocType2="Use Case">
         <Element Document=""> <State>{$item2/@name}</State>
              <IncomingTransition>{$t1/@xmi.idref}</IncomingTransition> </Element>
         <Element Document="">{$item3/../Title}</Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R35"  RuleType="refinement" DocType1="Statechart Diagram"
DocType2="Use Case">
   <Query>

declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

declare function local:getTransitioninState() as item()*
{
```

```
for $itemF in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Transition
return $itemF

};

declare function local:getStateinState($transition as node()) as item()
{
for $itemG in doc(Â£1Â£)//UML:SimpleState
where $itemG/UML:StateVertex.incoming/UML:Transition/@xmi.idref =
$transition/@xmi.idref
return $itemG

};

let $item1 := local:getTransitioninState()
for $t1 in $item1
for $item2 in local:getStateinState($t1)
for $item3 in doc(Â£2Â£)//Use_Case/Flow_of_events/Event
where
s:containsInDistance($item3, $item2/@name)

  </Query>
  <Action>
    <Relation RuleID="R35" Type="refinement" DocType1="Statechart Diagram"
              DocType2="Use Case">
      <Element Document=""> <State>{$item2/@name} </State>
          <IncomingTransition>{$t1/@xmi.idref}</IncomingTransition> </Element>
      <Element Document="">{$item3/../../Title} <Event/></Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R36"  RuleType="implements" DocType1="Statechart Diagram"
DocType2="Use Case">
  <Query>

declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

declare function local:getTransitioninState() as item()*
{
for $itemF in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Transition
return $itemF

};

declare function local:getStateinState($transition as node()) as item()
{
for $itemG in doc(Â£1Â£)//UML:SimpleState
where $itemG/UML:StateVertex.incoming/UML:Transition/@xmi.idref =
Stransition/@xmi.idref
```

```
return $itemG

};

let $item1 := local:getTransitioninState()
for $t1 in $item1
for $item2 in local:getStateinState($t1)
for $item3 in doc(Â£2Â£)//Use_Case/Flow_of_events/Event
where
s:containsInDistance($item3, $item2/@name)

  </Query>
  <Action>
    <Relation RuleID="R36" Type="implements" DocType1="Statechart Diagram"
              DocType2="Use Case">
      <Element Document=""> <State>{$item2/@name} </State>
          <IncomingTransition>{$t1/@xmi.idref}</IncomingTransition> </Element>
      <Element Document="">{$item3/../../Title}</Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R73"  RuleType="satisfiability" DocType1="Class Diagram"
DocType2="Use Case">
  <Query>

declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

for $item1 in doc(Â£1Â£)//UML:Classifier.feature/UML:Operation/@name
for $item2 in doc(Â£2Â£)//Use_Case/Description

let $t1 := $item1/../../../@name

where

s:containsInDistance($item2,$item1, $t1)
  </Query>
  <Action>
    <Relation RuleID="R73" Type="satisfiability" DocType1="Class Diagram"
              DocType2="Use Case">
      <Element Document=""> <Class>{$t1}</Class> <Operation>{$item1}</Operation>
      </Element>
      <Element Document="">{$item2/../Title}<Description/>
      </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R37"  RuleType="implements" DocType1="Class Diagram"
DocType2="Feature Model">
  <Query>

declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";
```

```
declare function local:getClassinClass($diagram as  xs:string) as item()*
{
for $itemE in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
where
$itemE/../../../../../../@name = $diagram
return $itemE
};

let $cl := local:getClassinClass(*1*)

for $item0 in $cl

for $item1 in
doc(Â£1Â£)//UML:Namespace.ownedElement/UML:Class/UML:Classifier.feature/UML:Operat
ion
for $item2 in doc(Â£2Â£)//Feature_Model/Feature
let $t1 := Sitem1/../../@name

where
$item1/../../@xmi.id = $item0/@xmi.idref
and
s:containsAInDistance($item2/Feature_name, $t1)
and
s:containsInDistance($item2/Description, $item1/@name)

   </Query>
   <Action>
      <Relation RuleID="R37" Type="implements" DocType1="Class Diagram"
                DocType2="Feature Model">
         <Element Document="">  <Class>{$t1}</Class>
                  <Operation>{$item1/@name}</Operation>  </Element>
         <Element Document="">{$item2/Feature_name}</Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R38"  RuleType="dependency" DocType1="Use Case"
DocType2="Feature Model">
   <Query>
declare namespace s="java:distanceControl.d";

declare function local:getChildrenFeature($parent as xs:string) as item()*
{
for $itemA in doc(Â£2Â£)//Feature/Relationship/Rel_feature

where
normalize-space($itemA/../../Feature_name)= normalize-space($parent)


return
$itemA
};
```

```
for
 $item1 in doc(Â£1Â£)//Use_Case/Description,
 $item2 in doc(Â£2Â£")//Feature_Model/Feature/Feature_name

where
local:getChildrenFeature(string($item2)) != ""
and
s:containsInDistance($item1,$item2, local:getChildrenFeature(string($item2)))

   </Query>
   <Action>
      <Relation RuleID="R38" Type="dependency" DocType1="Use Case"
                DocType2="Feature Model">
         <Element Document=""> {$item1/../Title} </Element>
         <Element Document="">{$item2}
            <Child>{ local:getChildrenFeature(string($item2))}</Child>
         </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R39"  RuleType="dependency" DocType1="Use Case"
DocType2="Feature Model">
   <Query>
declare namespace s="java:distanceControl.d";

declare function local:getParentFeature($child as xs:string) as item()
{
for $itemA in doc(Â£2Â£)//Relationship/Rel_feature

where
normalize-space($itemA)= normalize-space($child)
return

$itemA/../../Feature_name
};
for
 $item1 in doc(Â£1Â£)//Use_Case/Description,
 $item2 in doc(Â£2Â£")//Feature_Model/Feature/Feature_name

where
local:getParentFeature(string($item2)) != ""
and
s:containsInDistance($item1,$item2, local:getParentFeature(string($item2)))

   </Query>
   <Action>
      <Relation RuleID="R39" Type="dependency" DocType1="Use Case"
                DocType2="Feature Model">
         <Element Document=""> {$item1/../Title} </Element>
         <Element Document="">{$item2}
                  <Child>{ local:getChildrenFeature(string($item2))}</Child>
         </Element>
      </Relation>
```

```
   </Action>
</TraceRule>
<TraceRule RuleID="R40"  RuleType="satisfiability" DocType1="Process Model"
DocType2="Feature Model">
   <Query>
   declare namespace s="java:distanceControl.d";

declare function local:getParentFeature($child as xs:string) as item()
{
for $itemA in doc(Â£2Â£)//Relationship/Rel_feature

where
normalize-space($itemA)= normalize-space($child)
return

$itemA/../../Feature_name
};

for
 $item1 in doc(Â£1Â£)//Process_Model/Process/Description,
 $item2 in doc(Â£2Â£)//Feature_Model/Feature/Feature_name
where
local:getParentFeature(string($item2)) != ""
and
s:containsInDistance($item1,$item2, local:getParentFeature(string($item2)))

   </Query>
   <Action>
      <Relation RuleID="R40" Type="satisfiability" DocType1="Process Model"
                DocType2="Feature Model">
         <Element Document=""> {$item1/../Process_name} </Element>
         <Element Document="">{$item2} </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R41"  RuleType="satisfiability" DocType1="Process Model"
DocType2="Feature Model">
   <Query>
declare namespace s="java:distanceControl.d";

declare function local:getChildrenFeature($parent as xs:string) as item()*
{
for $itemA in doc(Â£2Â£)//Feature/Relationship/Rel_feature

where
normalize-space($itemA/../../Feature_name)= normalize-space($parent)


return
SitemA
};

for
 $item1 in doc(Â£1Â£)//Process_Model/Process/Description,
 $item2 in doc(Â£2Â£)//Feature_Model/Feature/Feature_name
```

```
where
local:getChildrenFeature(string($item2)) != ""
and
s:containsInDistance($item1,$item2, local:getChildrenFeature(string($item2)))
  </Query>
  <Action>
    <Relation RuleID="R41" Type="satisfiability" DocType1="Process Model"
              DocType2="Feature Model">
      <Element Document=""> {$item1/../Process_name} </Element>
      <Element Document="">{$item2} </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R42"  RuleType="satisfiability" DocType1="Module Model"
DocType2="Feature Model">
  <Query>
  declare namespace s="java:distanceControl.d";

declare function local:getParentFeature($child as xs:string) as item()
{
for $itemA in doc(Â£2Â£)//Relationship/Rel_feature

where
normalize-space($itemA)= normalize-space($child)
return

$itemA/../../Feature_name
};

for
 $item1 in doc(Â£1Â£)//Module_Model/Module/Description,
 $item2 in doc(Â£2Â£)//Feature_Model/Feature/Feature_name
where
local:getParentFeature(string($item2)) != ""
and
s:containsInDistance($item1,$item2, local:getParentFeature(string($item2)))

  </Query>
  <Action>
    <Relation RuleID="R42" Type="satisfiability" DocType1="Module Model"
              DocType2="Feature Model">
      <Element Document=""> {$item1/../Module_name} </Element>
      <Element Document="">{$item2} </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R43"  RuleType="containment" DocType1="Module Model"
DocType2="Class Diagram">
  <Query>
  declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

for $item1 in doc(Â£2Â£)//UML:Classifier.feature/UML:Operation/@name
for $item2 in doc(Â£1Â£")//Module/Description
```

```
let $t1 := $item1/../../../@name

where

s:containsInDistance($item2,$item1, $t1)


   </Query>
   <Action>
      <Relation RuleID="R43" Type="containment" DocType1="Module Model"
DocType2="Class Diagram">
         <Element Document="">{$item2/../Module_name} <Description/>
         </Element>
         <Element Document=""><Class>{$t1}</Class> <Operation> {$item1}</Operation>
         </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R86a"  RuleType="satisfiability" DocType1="Subsystem Model"
DocType2="Feature Model">
   <Query>


declare namespace s="java:distanceControl.d";

declare function local:getParentFeature($child as xs:string) as item()
{
for $itemA in doc(Â£2Â£)//Relationship/Rel_feature

where
normalize-space($itemA)= normalize-space($child)
return

$itemA/../../Feature_name
};

for
 $item1 in doc(Â£1Â£)//Subsystem/Description,
 $item2 in doc(Â£2Â£)//Feature_Model/Feature/Feature_name

where
normalize-space(local:getParentFeature(string($item2))) = ""
and
s:containsInDistance($item1,$item2)


   </Query>
   <Action>
      <Relation RuleID="R44" Type="satisfiability" DocType1="Subsystem Model"
                DocType2="Feature Model">
         <Element Document=""> {$item1/../Subsystem_name} </Element>
         <Element Document="">{$item2} </Element>
      </Relation>
   </Action>
</TraceRule>
```

```
<TraceRule RuleID="R45"  RuleType="satisfiability" DocType1="Process Model"
DocType2="Feature Model">
   <Query>


   declare namespace s="java:distanceControl.d";

declare function local:getParentFeature($child as xs:string) as item()
{
for $itemA in doc(Â£2Â£)//Relationship/Rel_feature

where
normalize-space($itemA)= normalize-space($child)
return

$itemA/../../Feature_name
};

for
 $item1 in doc(Â£1Â£)//Process_Model/Process/Description,
 $item2 in doc(Â£2Â£)//Feature_Model/Feature/Feature_name

where

normalize-space(local:getParentFeature(string($item2))) = ""
and
s:containsInDistance($item1,$item2)

   </Query>
   <Action>
      <Relation RuleID="R45" Type="satisfiability" DocType1="Process Model"
DocType2="Feature Model">
         <Element Document=""> {$item1/../Process_name} </Element>
         <Element Document="">{$item2} </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R46"  RuleType="satisfiability" DocType1="Module Model"
DocType2="Feature Model">
   <Query>
   declare namespace s="java:distanceControl.d";

declare function local:getParentFeature($child as xs:string) as item()
{
for $itemA in doc(Â£2Â£)//Relationship/Rel_feature

where
normalize-space($itemA)= normalize-space($child)
return

$itemA/../../Feature_name
};

for
 $item1 in doc(Â£1Â£)//Module_Model/Module/Description,
```

```
 $item2 in doc(Â£2Â£)//Feature_Model/Feature/Feature_name
where

normalize-space(local:getParentFeature(string($item2))) = ""
and
s:containsInDistance($item1,$item2)

   </Query>
   <Action>
      <Relation RuleID="R46" Type="satisfiability" DocType1="Module Model"
                DocType2="Feature Model">
         <Element Document=""> {$item1/../Module_name} </Element>
         <Element Document="">{$item2} </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R47" RuleType="dependency" DocType1="Class Diagram"
DocType2="Feature Model">
   <Query>


declare namespace UML="org.omg.xmi.namespace.UML";

declare namespace s="java:distanceControl.d";

declare function local:getClassinClass($diagram as  xs:string) as item()*
{
for $itemE in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
where
SitemE/../../../../../../@name = Sdiagram
return $itemE
};

declare function local:getParentFeature($child as xs:string) as item()
{
for $itemA in doc(Â£2Â£)//Relationship/Rel_feature

where
normalize-space($itemA)= normalize-space($child)
return

$itemA/../../Feature_name
};

let $cl := local:getClassinClass(*1*)

for $item0 in $cl

for $item1 in
doc(Â£1Â£)//UML:Namespace.ownedElement/UML:Class/UML:Classifier.feature/UML:Operat
ion
for $item2 in doc(Â£2Â£)//Feature_Model/Feature/Description
```

```
let $t1 := $item1/../../@name

where
$item1/../../@xmi.id = $item0/@xmi.idref
and
s:containsInDistance($item2, $item1/@name, $t1)
and
normalize-space(local:getParentFeature(string($item2/../Feature_name))) != ""


  </Query>
  <Action>
    <Relation RuleID="R47" Type="dependency" DocType1="Class Diagram"
DocType2="Feature Model">
      <Element Document=""> <Class>{$t1}</Class>
                  <Operation>{$item1/@name}</Operation> </Element>
      <Element Document="">
            {local:getParentFeature(string($item2/../Feature_name))}</Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R48" RuleType="dependency" DocType1="Sequence Diagram"
DocType2="Class Diagram">
  <Query>
declare namespace UML="org.omg.xmi.namespace.UML";

declare function local:getMessageinSeq() as item()*
{
for $itemA in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Link
return $itemA

};

declare function local:getObjectinSeq($link as node()) as item()
{
for $itemB in doc(Â£1Â£)//UML:Link
where $itemB/@xmi.id = $link/@xmi.idref

return $itemB/UML:Link.connection/UML:LinkEnd/UML:LinkEnd.instance/UML:Object
};

declare function local:getObjectinModel($object as node()) as item()*
{

for $itemC in doc(Â£1Â£)//UML:Object
where $itemC/@xmi.id = $object/@xmi.idref

return $itemC/UML:Instance.classifier/UML:Class

};

declare function local:getClassObjectinSeq($class as node()) as item()
{
```

```
for $itemD in doc(Â£1Â£)//UML:Class
where $itemD/@xmi.id = $class/@xmi.idref

return $itemD

};

declare function local:getClassname2($class as node()) as item()*
{
for $itemD2 in doc(Â£2Â£)//UML:Class
where $itemD2/@name = $class/@name

return $itemD2
};

declare function local:getClassinClass($class as node(), $diagram as  xs:string) as item()*
{
for $itemE in
doc(Â£2Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
where $itemE/@xmi.idref = $class/@xmi.id
and $itemE/../../../../../../@name = $diagram
return $itemE
};

let $item1 := local:getMessageinSeq()
for $t1 in $item1
for $item2 in local:getObjectinSeq($t1)
for $item3 in local:getObjectinModel($item2)
for $item4 in local:getClassObjectinSeq($item3)
for $item4_1 in local:getClassname2($item4)
for $item5 in local:getClassinClass($item4_1,*2*)
where
$t1/../../../../../../@name = *1*
and
local:getParentClass($item4_1/@name) != ""


  </Query>
  <Action>
    <Relation  RuleID="R48" Type="dependency" DocType1="Sequence Diagram"
DocType2="Class Diagram">
      <Element Document="">{$t1/../../../../../../@xmi.id} {$t1/../../../../../../@name} {$t1}
{$item2} {$item3}</Element>
      <Element Document="">{local:getParentClass($item4_1/@name)}</Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R49" RuleType="evoluation" DocType1="Use Case" DocType2="Use
Case">
  <Query>
  declare namespace s="java:distanceControl.d";

  for $item1 in doc(Â£1Â£)//Use_Case/Title,
```

```
      $item2 in doc(Â£2Â£)//Use_Case/Title

   where
      $item1/../@UseCaseID != $item2/../@UseCaseID and
      $item1/../@System = $item1/../@System and
      $item1/../@Product_Member = $item2/../@Product_Member and
      s:containsInDistance($item1, $item2)
   </Query>
   <Action>
      <Relation RuleID="R49" Type="evoluation" DocType1="Use Case"
                DocType2="Use Case">
         <Element Document=""> {$item1/../@UseCaseID} {$item1}
            <System>{$item1/../@System} {$item1/../@Product_Member} </System>
         </Element>
         <Element Document=""> {$item2/../@UseCaseID} {$item2}
            <System>{$item2/../@System} {$item2/../@Product_Member} </System>
         </Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R50" RuleType="evoluation" DocType1="Class Diagram"
DocType2="Class Diagram">
   <Query>

   declare namespace UML="org.omg.xmi.namespace.UML";

   declare namespace s="java:distanceControl.d";

   declare function local:getClassinClass($diagram as  xs:string) as item()*
   {
      for $itemE in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
      where
          $itemE/../../../../../../@name = $diagram
      return $itemE
   };


   let $cl := local:getClassinClass(*1*)
   let $cl2 := local:getClassinClass(*2*)

   for $item0 in $cl
   for $item00 in $cl2

   for $item1 in doc(Â£1Â£)//UML:Class
   for $item2 in doc(Â£2Â£)//UML:Class


   where
      ($item1/@xmi.id = $item0/@xmi.idref
   and
      $item2/@xmi.id = $item00/@xmi.idref
   and
```

```
      s:containsInDistance($item2/@name, $item1/@name))
   </Query>
   <Action>
      <Relation RuleID="R50" Type="evoluation" DocType1="Class Diagram"
                DocType2="Class Diagram">
         <Element Document="">{$item1}</Element>
         <Element Document="">{$item2}</Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R51" RuleType="dependency" DocType1="Class Diagram"
DocType2="Use Case">
   <Query>
   declare namespace UML="org.omg.xmi.namespace.UML";

   declare namespace s="java:distanceControl.d";

   for $itemA in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class

   for $item1 in
doc(Â£1Â£)//UML:Package/UML:Namespace.ownedElement/UML:Class/UML:Classifier.featu
re/UML:Operation/@name
   for $item2 in doc(Â£2Â£)//Use_Case

   let $t1 := $item1/../../../@name

   where
      $itemA/../../../../../parent::node()/@name = *1*
   and
      $item1/../../../@xmi.id = $itemA/@xmi.idref
   and
      s:containsInDistance($item2/Title, $t1)
   and
      s:containsInDistance($item2/Description, $item1)

   </Query>
   <Action>
      <Relation RuleID="R51" Type="dependency" DocType1="Class Diagram"
DocType2="Use Case">
         <Element Document=""><Classs>{$t1}</Class> </Element>
         <Element Document="">{$item2/Title}</Element>
      </Relation>
   </Action>
</TraceRule>
<TraceRule RuleID="R52" RuleType="dependency" DocType1="Statechart Diagram"
DocType2="Use Case">
   <Query>


   declare namespace UML="org.omg.xmi.namespace.UML";

   declare namespace s="java:distanceControl.d";
```

```
  declare function local:getTransitioninState() as item()*
  {
    for $itemF in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Transition
   return $itemF

  };

  declare function local:getStateinState($transition as node()) as item()
  {
  for $itemG in doc(Â£1Â£)//UML:SimpleState
  where $itemG/UML:StateVertex.incoming/@xmi.idref = $transition/@xmi.id

  return $itemG

  };

  let $item1 := local:getTransitioninState()
  for $t1 in $item1
  for $item2 in local:getStateinState($t1)

  for $item3 in doc(Â£2Â£)//Use_Case/Description

  for $itemE in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class

  where $itemE/@name =$item2/../parent::node()/@name
    and $itemE/../../../../../parent::node()/@name = *1*
    and s:containsInDistance($item3, $item2/@name)

  </Query>
  <Action>
    <Relation RuleID="R52" Type="dependency" DocType1="Statechart Diagram"
              DocType2="Use Case">
      <Element Document=""> {$item2} </Element>
      <Element Document=""> {$item3/../Title} </Element>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R53" RuleType="dependency" DocType1="Sequence Diagram"
DocType2="Use Case">
  <Query>


  declare namespace UML="org.omg.xmi.namespace.UML";

  declare namespace s="java:distanceControl.d";

  declare function local:getMessageinSeq() as item()*
  {
```

```
        for $itemA in
doc(Â£1Â£)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml
1SemanticModelBridge.element/UML:Link
        return $itemA

   };

   declare function local:getObjectinSeq($link as node()) as item()
   {
    for $itemB in doc(Â£1Â£)//UML:Link
    where $itemB/@xmi.id = $link/@xmi.idref

    return $itemB/UML:Link.connection/UML:LinkEnd/UML:LinkEnd.instance/UML:Object
   };

   declare function local:getObjectinModel($object as node()) as item()*
   {

    for $itemC in doc(Â£1Â£)//UML:Object
   where $itemC/@xmi.id = $object/@xmi.idref

   return $itemC/UML:Instance.classifier/UML:Class

   };

   declare function local:getClassObjectinSeq($class as node()) as item()
   {
    for $itemD in doc(Â£1Â£)//UML:Class
   where $itemD/@xmi.id = $class/@xmi.idref

   return $itemD

   };

   declare function local:getClassinClass($class as node(), $diagram as xs:string)as item()*
   {
      for $itemE in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UML:GraphElem
ent.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1SemanticModelBridge.elemen
t/UML:Class
      where $itemE/@name =$class/@name
      and $itemE/../../../../../parent::node()/@name = $diagram
      return $itemE
   };

   let $item1 := local:getMessageinSeq()
   for $t1 in $item1
   for $item2 in local:getObjectinSeq($t1)
   for $item3 in local:getObjectinModel($item2)
   for $item4 in local:getClassObjectinSeq($item3)
   for $item5 in local:getClassinClass($item4, *1*)

   for $item6 in doc(Â£2Â£)//Use_Case/Description
```

```
  where s:containsInDistance($item6, $item5)

  </Query>
  <Action>
    <Relation RuleID="R53" Type="dependency" DocType1="Sequence Diagram"
DocType2="Use Case">
      <Element Document=""> {$item5} </Element>
      <Element Document=""> {$item6/../Title}  </Element>
    </Relation>
  </Action>
</TraceRule>
```

B.2. Indirect Traceability Rules

```
<TraceRule RuleID="R54" RuleType="similar" DocType1="XML-Based-Rel"
DocType2="XML-Based-Rel">
  <Query>

for $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="implements"],
$item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="implements"]

where $item1/@DocType1="Use Case" and $item1/@DocType2 ="Sequence Diagram"
and $item2/@DocType1="Use Case" and $item2/@DocType2="Sequence Diagram"
and (string($item1/Element[2]/Link) = string($item2/Element[2]/Link))
and ($item1/Element[1]/@Document != $item2/Element[1]/@Document)
and ($item1/Element[2]/@Document = $item2/Element[2]/@Document)

  </Query>
  <Action>
    <Relation RuleID="R54" Type="similar" Term="sequence diagram implements use case">
      <Element>{$item1/Element[1]/@Document} {$item1/Element[1]/Title}</Element>
      <Element>{$item2/Element[1]/@Document}{$item2/Element[1]/Title}</Element>
      <Implements>{$item1/Element[2]/@Document}{$item1/Element[2]/Link}
</Implements>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R55" RuleType="different" DocType1="XML-Based-Rel"
DocType2="XML-Based-Rel">
  <Query>

declare function local:getParentFeature($child as xs:string) as item()
{
for $itemA in
doc(Â£2Â£)//Relationship/Rel_feature
where normalize-space($itemA)= normalize-space($child)
return $itemA/../../Feature_name
};

declare function local:getParentofVariantFeatures($one as node(), $two as node())as item()
{
for $item1 in doc(Â£2Â£)//Feature,
$item2 in doc(Â£2Â£)//Feature
where (normalize-space($item1/Feature_name) = normalize-space($one)
and normalize-space($item2/Feature_name) = normalize-space($two)
and local:getParentFeature($item1/Feature_name) =
local:getParentFeature($item2/Feature_name)
and local:getParentFeature($item1/Feature_name) != ""
and normalize-space($item1/Feature_name) != normalize-space($item2/Feature_name)
and $item1/Existential = $item2/Existential
and ($item1/Existential = "Optional" or $item1/Existential = "Alternative"))
return true()
};

for $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="containment"],
 $item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="containment"]
```

```
where $item1/@DocType1 ="Use Case" and $item1/@DocType2="Feature Model"
and $item2/@DocType1="Use Case" and $item2/@DocType2="Feature Model"
and (string($item1/Element[1]/@Document) != string($item2/Element[1]/@Document))
and ($item1/Element[2]/@Document = $item2/Element[2]/@Document)
and ($item1/Element[2]/Feature_name != $item2/Element[2]/Feature_name) and
local:getParentofVariantFeatures($item1/Element[2]/Feature_name,$item2/Element[2]/Feature_n
ame)

  </Query>
  <Action>
    <Relation RuleID="R55" Type="different" Term="use case contains feature model">
      <Element>{$item1/Element[1]/@Document}{$item1/Element[1]/Title}</Element>
      <Element>{$item1/Element[1]/@Document}{$item2/Element[1]/Title}</Element>
      <Containment>{$item1/Element[2]/Feature_name}</Containment>
      <Containment>{$item2/Element[2]/Feature_name}</Containment>
      <VariantOf>{local:getParentFeature($item1/Element[2]/Feature_name)}
                  </VariantOf>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R56" RuleType="similar" DocType1="XML-Based-Rel"
DocType2="XML-Based-Rel">
  <Query>
declare namespace UML="org.omg.xmi.namespace.UML";

for $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="satisfiability"],
$item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="satisfiability"]
where
$item1/@DocType1 ="Class Diagram" and $item1/@DocType2="Use Case"  and
$item2/@DocType1="Class Diagram" and $item2/@DocType2="Use Case"
and
(string($item1/Element[1]/Class) = string($item2/Element[1]/Class))
and ($item1/Element[1]/@Document = $item2/Element[1]/@Document)
and ($item1/Element[2]/@Document != $item2/Element[2]/@Document)



  </Query>
  <Action>
    <Relation RuleID="R56" Type="similar" Term="class diagram satisfies use case">
      <Element>{$item1/Element[2]/@Document} {$item1/Element[2]/Title}</Element>
      <Element>{$item2/Element[2]/@Document} {$item2/Element[2]/Title}</Element>
      <Satisfiability>{$item1/Element[1]/@Document} {$item1/Element[1]/Class}
</Satisfiability>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R57" RuleType="similar" DocType1="XML-Based-Rel"
DocType2="XML-Based-Rel">
  <Query>
declare namespace UML="org.omg.xmi.namespace.UML";

for $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="implements"],
$item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="implements"]
where
```

```
$item1/@DocType1 ="Class Diagram" and $item1/@DocType2="Use Case"  and
$item2/@DocType1="Class Diagram" and $item2/@DocType2="Use Case"
and
(string($item1/Element[1]/Class) = string($item2/Element[1]/Class))
and ($item1/Element[1]/@Document = $item2/Element[1]/@Document)
and ($item1/Element[2]/@Document  != $item2/Element[2]/@Document)


  </Query>
  <Action>
    <Relation RuleID="R57" Type="similar" Term="class diagram implements use case">
      <Element>{$item1/Element[2]/@Document} {$item1/Element[2]/Title}</Element>
      <Element>{$item2/Element[2]/@Document} {$item2/Element[2]/Title}</Element>
      <Implements>{$item1/Element[1]/@Document} {$item1/Element[1]/Class}
                              </Implements>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R58" RuleType="similar" DocType1="XML-Based-Rel"
DocType2="XML-Based-Rel">
  <Query>

declare namespace UML="org.omg.xmi.namespace.UML";

for $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="satisfiability"],
$item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="satisfiability"]
where
$item1/@DocType1 ="Class Diagram" and $item1/@DocType2="Use Case"  and
$item2/@DocType1="Class Diagram" and $item2/@DocType2="Use Case"
and
(string($item1/Element[1]/Class) = string($item2/Element[1]/Class))
and ($item1/Element[1]/@Document = $item2/Element[1]/@Document)
and (string($item1/Element[2]/@Document)  != string($item2/Element[2]/@Document))
and (string($item1/Element[2]/Title)  != "")
and (string($item2/Element[2]/Title) != "")

</Query>
  <Action>
    <Relation RuleID="R58" Type="similar" Term="class diagram satisfies use case">
      <Element>{$item1/Element[2]/@Document} {$item1/Element[2]/Title}</Element>
      <Element>{$item2/Element[2]/@Document} {$item2/Element[2]/Title}</Element>
      <Satisfiability>{$item1/Element[1]/@Document} {$item1/Element[1]/Class}
</Satisfiability>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R59" RuleType="similar" DocType1="XML-Based-Rel"
DocType2="XML-Based-Rel">
  <Query>


declare namespace UML="org.omg.xmi.namespace.UML";

for $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="dependency"],
$item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="dependency"]
```

```
where
$item1/@DocType1 ="Sequence Diagram" and $item1/@DocType2="Use Case"  and
$item2/@DocType1="Sequence Diagram" and $item2/@DocType2="Use Case"
and
(string($item1/Element[2]/Link) = string($item2/Element[2]/Link))
and ($item1/Element[2]/@Document = $item2/Element[2]/@Document)
and ($item1/Element[1]/@Document  != $item2/Element[1]/@Document)
and (string($item1/Element[1]/Title)  != "")
and (string($item2/Element[1]/Title) != "")
and $item1/e/UML:Link/@xmi.idref = $item2/e/UML:Link/@xmi.idref

  </Query>
  <Action>
    <Relation RuleID="R59" Type="similar" Term="sequence diagram depends on use case">
      <Element>{$item1/Element[1]/@Document} {$item1/Element[1]/Title}</Element>
      <Element>{$item2/Element[1]/@Document} {$item2/Element[1]/Title}</Element>
      <Implements>{$item1/Element[2]/@Document} {$item1/Element[2]/Link}
                                                              </Implements>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R60" RuleType="similar" DocType1="XML-Based-Rel"
DocType2="XML-Based-Rel">
  <Query>

declare namespace UML="org.omg.xmi.namespace.UML";

for $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="refinement"],
$item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="refinement"]
where
$item1/@DocType1 ="Statechart Diagram" and $item1/@DocType2="Sequence Diagram"  and
$item2/@DocType1="Statechart Diagram" and $item2/@DocType2="Sequence Diagram"
and
(string($item1/Element[1]/Object) = string($item2/Element[1]/Object))
and (string($item1/Element[1]/Operation) = string($item2/Element[1]/Operation))
and ($item1/Element[1]/@Document = $item2/Element[1]/@Document)
and ($item1/Element[2]/@Document  != $item2/Element[2]/@Document)

  </Query>
  <Action>
    <Relation RuleID="R60" Type="similar" Term="statechart diagram refines sequence
diagram">
      <Element>{$item1/Element[2]/@Document}
               {$item1/Element[1]/Object}{$item1/Operation} </Element>
      <Element>{$item2/Element[2]/@Document}
               {$item2/Element[1]/Object}{$item2/Operation} </Element>
      <Implements>{$item1/Element[1]/@Document}
                                  {$item1/Element[1]/State} </Implements>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R61" RuleType="similar" DocType1="XML-Based-Rel"
DocType2="XML-Based-Rel">
  <Query>
```

```
declare namespace UML="org.omg.xmi.namespace.UML";

for $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="refinement"],
$item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="refinement"]
where
$item1/@DocType1 ="Sequence Diagram" and $item1/@DocType2="Class Diagram"  and
$item2/@DocType1="Sequence Diagram" and $item2/@DocType2="Class Diagram"
and
(string($item1/Element[1]/Object) = string($item2/Element[1]/Object))
and ($item1/Element[1]/@Document = $item2/Element[1]/@Document)
and ($item1/Element[2]/@Document  != $item2/Element[2]/@Document)


  </Query>
  <Action>
    <Relation RuleID="R61" Type="similar" Term="sequence diagram refines class diagram">
      <Element>{$item1/Element[2]/@Document} {$item1/Element[2]/Class}</Element>
      <Element>{$item2/Element[2]/@Document} {$item2/Element[2]/Class}</Element>
      <Refinement>{$item1/Element[1]/@Document} {$item1/Element[1]/Object}
                                                                    </Refinement>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R62" RuleType="similar" DocType1="XML-Based-Rel"
DocType2="XML-Based-Rel">
  <Query>

declare namespace UML="org.omg.xmi.namespace.UML";


for $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="dependency"],
$item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="dependency"]
where
$item1/@DocType1 ="Class Diagram" and $item1/@DocType2="Use Case"  and
$item2/@DocType1="Class Diagram" and $item2/@DocType2="Use Case"
and
(string($item1/Element[1]/Class) = string($item2/Element[1]/Class))
and ($item1/Element[2]/@Document != $item2/Element[2]/@Document)
and ($item1/Element[1]/@Document = $item2/Element[1]/@Document)



  </Query>
  <Action>
    <Relation RuleID="R62" Type="similar" Term="class diagram implements use case">
      <Element>{$item1/Element[2]/@Document} {$item1/Element[2]/Title}</Element>
      <Element>{$item2/Element[2]/@Document} {$item2/Element[2]/Title}</Element>
      <Implements>{$item1/Element[1]/@Document} {$item1/Element[1]/Class}
                                                    </Implements>
    </Relation>
  </Action>
</TraceRule>
<TraceRule RuleID="R63" RuleType="different" DocType1="XML-Based-Rel"
DocType2="XML-Based-Rel">
  <Query>
```

```
declare namespace UML="org.omg.xmi.namespace.UML";
declare function local:getParentClass($child as xs:string) as item()
{
for $itemA in doc(Â£1Â£)//UML:Generalization/UML_Generalization.child

where
$itemA/UML:Class/@xmi.idref = $child
return

$itemA/../UML:Generalization.parent/UML:Class
};

declare function local:getParentofVariantClasses($one as xs:string, $two as xs:string)as item()
{    for $item1 in doc(Â£2Â£)//UML:Generalization/UML:Generalization.child,
     $item2 in doc(Â£2Â£)//UML:Generalization/UML:Generalization.child
where
($item1/UML:Class/@xmi.idref = $one and
 $item2/UML:Class/@xmi.idref = $two and
 local:getParentClass($item1/UML:Class/@xmi.idref ) =
 local:getParentClass($item2/UML:Class/@xmi.idref) and
 local:getParentClass($item1/UML:Class/@xmi.idref) != "" and
 $item1/UML:Class/@xmi.idref != $item2/UML:Class/@xmi.idref )
return local:getParentClass($item1/UML:Class/@xmi.idref)
};
declare function local:getClassID($name as xs:string)as xs:string
{    for $itemB in doc(Â£2Â£)//UML:Class/@name
where  $itemB = $name
return  $itemB/../@xmi.id
};
for $item1 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="implements"],
$item2 in doc("file:///c:/Direct_TraceRel.xml")//Relation[@Type="implements"]
where
$item1/@DocType1 ="Class Diagram" and $item1/@DocType2="Feature Model"  and
$item2/@DocType1="Class Diagram" and $item2/@DocType2="Feature Model"
and
(string($item1/Element[2]/Feature_name) = string($item2/Element[2]/@Feature))
and ($item1/Element[1]/@Document = $item2/Element[1]/@Document)
and ($item1/Element[1]/Class != $item2/Element[1]/Class)
and
local:getParentofVariantClasses(local:getClassID($item1/Element[1]/Class/@name),local:getCla
ssID($item2/Element[1]/Class/@name))

  </Query>
  <Action>
    <Relation RuleID="R63" Type="different" Term="class implements feature">
      <Element>{$item1/Element[2]/Feature_name}</Element>
      <Element>{$item2/Element[2]/Feature_name} </Element>
      <Implements>{$item1/Element[1]/Class}</Implements>
      <Implements>{$item2/Element[1]/Class}</Implements>
      <VariantOf>{local:getParentClass(local:getClassID($item1/Element[1]/Class/@name))}
                                                    </VariantOf>
    </Relation>
  </Action>
</TraceRule>
```

Appendix C – Extended XQUERY Functions

C.1.getTransitioninState

```
declare function local:getTransitioninState() as item()*
{
for $itemF in
doc(£££)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBri
dge/UML:Uml1SemanticModelBridge.element/UML:Transition
return $itemF
};
```

C.2. getStateinState

```
declare function local:getStateinState($transition as node()) as
item()
{
for $itemG in doc(£££)//UML:SimpleState
where
$itemG/UML:StateVertex.incoming/UML:Transition/@xmi.idref =
$transition/@xmi.idref
return $itemG
};
```

C.3. getMessageinSeq

```
declare function local:getMessageinSeq() as item()*
{
for $itemA in
doc(£££)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBri
dge/UML:Uml1SemanticModelBridge.element/UML:Link
return $itemA
};
```

C.4. getObjectinSeq

```
declare function local:getObjectinSeq($link as node()) as item()
{
for $itemB in doc(£££)//UML:Link
where $itemB/@xmi.id = $link/@xmi.idref
return
$itemB/UML:Link.connection/UML:LinkEnd/UML:LinkEnd.instance/UML:O
bject
};
```

C.5. getClassID

```
declare function local:getClassname($name as xs:string) as
xs:string
{
for $itemB in doc(£££)//UML:Class/@name
where $itemB = $name
return $itemB/../@xmi.id
};
```

C.6. getClassObjectinSeq

```
declare function local:getClassObjectinSeq($diagram as xs:string)
as item()
{
for $itemE in
doc(£££)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/UM
L:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Uml1
SemanticModelBridge.element/UML:Class
where  $itemE/../../../../../../@name = $diagram
return $itemE
};
```

C.7. getParentFeature

```
declare function local:getParentFeature($child as xs:string) as
item()
{
for $itemA in doc(£££)//Relationship/Rel_feature

where
normalize-space($itemA)= normalize-space($child)
return

$itemA/../../Feature_name
};
```

C.8. getChildrenFeature

```
declare function local:getChildrenFeature($parent as xs:string)
as item()*
{
for $itemA in doc(£££)//Feature/Relationship/Rel_feature
where
normalize-space($itemA/../../Feature_name)= normalize-
space($parent)
return
$itemA
};
```

C.9. getFeatureofSubsystem

```
declare function local:getFeatureofSubsystem($subsystem as
xs:string) as item()*
{
for $itemA in
doc(£££)//Feature_Model/Feature/Allocated_to_Subsystem
where
normalize-space($itemA)= normalize-space($subsystem)
return $itemA/$itemA/../Feature_name
};
```

C.10. getOperationinSeq

```
declare function local:getOperationinSeq() as item()*
{
for $itemA in
doc(£££)//UML:GraphElement.semanticModel/UML:Uml1SemanticModelBri
dge/UML:Uml1SemanticModelBridge.element/UML:Operation
return $itemA
};
```

C.11. getOperationinModel

```
declare function local:getOperationinModel($operation as node())
as item()*
{
 for $itemC in doc(£££)//UML:Classifier.feature/UML:Operation
where $itemC/@xmi.id = $operation/@xmi.idref
return $itemC
};
```

C.12. getStateofOperationinState

```
declare function local:getStateofOperationinState($operation as
node()) as item()
{
for $itemD in doc(£££)//UML:SimpleState
where $itemD/@name = $operation/@name
return $itemD
};
```

C.13. getClassinClass

```
declare function local:getClassinClass($diagram as  xs:string) as
item()*
{
for $itemE in
doc(Â£1Â£)//UML:Diagram/UML:GraphElement.contained/UML:GraphNode/
UML:GraphElement.semanticModel/UML:Uml1SemanticModelBridge/UML:Um
l1SemanticModelBridge.element/UML:Class
where
$itemE/../../../../../../@name = $diagram
return $itemE
};
```

C.14. getParentofVariantClasses

```
declare function local:getParentofVariantClasses($one as node(),
$two as node())as item()
{
for $item1 in
doc(Â£2Â£)//UML:Generalization/UML_Generalization.child,
      $item2 in
doc(Â£2Â£)//UML:Generalization/UML_Generalization.child
where

where
($item1/UML:Class/@xmi.idref = $one
and
$item1/UML:Class/@xmi.idref = $two
and local:getParentClass($item1/UML:Class/@xmi.idref) =
local:getParentClass($item2/UML:Class/@xmi.idref)
and local:getParentClass($item1/UML:Class/@xmi.idref) != "" and
$item1/UML:Class/@xmi.idref != $item2/UML:Class/@xmi.idref)
return local:getParentClass($item1/UML:Class/@xmi.idref)
};
```

C.15. getParentofVariantFeatures

```
declare function local:getParentofVariantFeatures($one as node(),
$two as node())as item()
{
for $item1 in doc(Â£2Â£)//Feature,
      $item2 in doc(Â£2Â£)//Feature
where (normalize-space($item1/Feature_name) = normalize-
space($one)
and normalize-space($item2/Feature_name) = normalize-space($two)
and local:getParentFeature($item1/Feature_name) =
local:getParentFeature($item2/Feature_name)
and local:getParentFeature($item1/Feature_name) != ""
and normalize-space($item1/Feature_name) != normalize-
space($item2/Feature_name)
and $item1/Existential = $item2/Existential
and ($item1/Existential = "Optional" or $item1/Existential =
"Alternative"))
return local:getParentFeature($item1/Feature_name)
};
```

C.16. getParentClass

```
declare function local:getParentClass($child as xs:string) as
item()
{
for $itemA in
doc(Â£1Ã£)//UML:Generalization/UML_Generalization.child

where
$itemA/UML:Class/@xmi.idref = $child
return

$itemA/../UML:Generalization.parent/UML:Class
};
```

Appendix D – Example Documents in Mobile-Phone Systems

D.1. Use Case – PM1

```
<?xml version="1.0" encoding="UTF-8"?>
<Use_Case UseCaseID="UC1" System="MobilePhone" Product_Member="PM1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Z:\web\XTraQue\Use_case.xsd">
        <Title>
                <VVG>Sending</VVG>
                <AT0>a</AT0>
                <NN1>Message</NN1>
        </Title>
        <Description>
                <AT0>The</AT0><NN1>phone</NN1><VBZ>is</VBZ><AJ0>able</AJ0>
                <TO0>to</TO0><VVI>send</VVI><AT0>a</AT0><NN1>text</NN1>
                <NN1>message</NN1><SC>.</SC><AT>The</AT><NN1>user</NN1>
                <VM>can</VM><VVI>specify</VVI><AT1>an</AT1><NN1>address</NN1>
                <IO>of</IO><AT1>a</AT1><NN1>receiver</NN1><II>by</II>
                <VVG>selecting</VVG><II>from</II><AT1>a</AT1><NN1>list</NN1>
                <IO>of</IO><NN2>contacts</NN2><SC>.</SC>
        </Description>
        <Level> Primary task</Level>
        <Preconditions>
                <AT>The</AT><NN1>user</NN1><VHZ>has</VHZ><VHZ>already</VHZ>
                <VVN>selected</VVN><NN1>function</NN1><IO>of</IO>
                <VVG>sending</VVG><AT1>a</AT1><NN1>text</NN1>
                <NN1>message</NN1><II>from</II><AT>the</AT><JJ>main</JJ>
                <NN1>menu</NN1><SC>.</SC>
        </Preconditions>
        <Postconditions>
                <AT>The</AT><NN1>phone</NN1><VHZ>has</VHZ><VVN>sent</VVN>
                <AT>the</AT><NN1>message</NN1><SC>.</SC>
        </Postconditions>
        <Primary_actor> The user </Primary_actor>
        <Secondary_actors/>
        <Flow_of_events>
                <Event>
                        <AT>The</AT><NN1>system</NN1><VVZ>shows</VVZ>
                        <AT1>an</AT1><NN1>editor</NN1><IF>for</IF>
                        <VVG>writing</VVG><AT1>a</AT1><NN1>message</NN1>
                        <SC>.</SC>
                </Event>
                <Event>
                        <AT>The</AT><NN1>user</NN1><NN1>key-in</NN1>
                        <AT1>a</AT1><NN1>phone</NN1><NN1>number</NN1>
                        <IO>of</IO><AT1>a</AT1><NN1>receiver</NN1>
                        <SC>.</SC><RR>Moreover</RR><SC>,</SC><AT>the</AT>
                        <NN1>user</NN1><VVI>select</VVI><AT1>a</AT1>
                        <NN1>phone</NN1><NN1>number</NN1><IO>of</IO>
                        <AT1>a</AT1><NN1>receiver</NN1><II>by</II>
                        <VVG>selecting</VVG><II>from</II><AT1>a</AT1>
                        <NN1>list</NN1><IO>of</IO><NN2>contacts</NN2>
```

```
...
        <SC>.</SC><VV0>Note</VV0><AT>the</AT><NN1>user</NN1>
        <VVI>send</VVI><AT>the</AT><NN1>text</NN1>
        <NN1>message</NN1><II>to</II><JJ>multiple</JJ>
        <NN2>receivers</NN2><II>by</II><VVG>inserting</VVG>
        <JJ>multiple</JJ><JJ>mobile</JJ><NN1>phone</NN1>
        <NN2>numbers</NN2><SC>.</SC>
</Event>
<Event>
        <AT>The</AT><NN1>system</NN1><VVD>displayed</VVD>
        <AT>the</AT><NN1>phone</NN1><NN1>number</NN1>
        <SC>.</SC>
</Event>
<Event>
        <AT>The</AT><NN1>user</NN1><VVI>enter</VVI>
        <AT>the</AT><NN1>message</NN1><SC>.</SC>
        <RR>Otherwise</RR><DD1>This</DD1><VBZ>is</VBZ>
        <VVN>limited</VVN><II>under</II><AT>the</AT>
        <JJ>maximum</JJ><NN1>size</NN1><IO>of</IO>
        <VVG>sending</VVG><AT>the</AT><NN1>text</NN1>
        <NN1>message</NN1><SC>.</SC>
</Event>
<Event>
        <AT>The</AT><NN1>system</NN1><VVD>displayed</VVD>
        <AT>the</AT><NN1>message</NN1><SC>.</SC>
        <VV0>Note</VV0><CST>that </CST><AT>the</AT>
        <NN2>events</NN2><VBR>are</VBR><XX>not</XX>
        <JJ>sequential</JJ><NN2>processes</NN2><SC>.</SC>
</Event>
<Event>
        <AT>The</AT><NN1>user</NN1><VVZ>confirms</VVZ>
        <VVG>sending</VVG><AT>the</AT><NN1>message</NN1>
        <SC>.</SC>
</Event>
<Event>
        <AT>The</AT><NN1>system</NN1><VVZ>establishes</VVZ>
        <AT>the</AT><NN1>connection</NN1><IF>for</IF>
        <VVG>sending</VVG><SC>.</SC>
</Event>
<Event>
        <CS>If </CS><AT>the</AT><NN1>connection</NN1>
        <VBZ>is</VBZ><RR>properly </RR><VVN>set</VVN>
        <SC>,</SC><AT>the</AT><NN1>system</NN1>
        <VVZ>sends</VVZ><AT>the</AT><NN1>message</NN1>
        <SC>.</SC><RR>Otherwise</RR><SC>,</SC>
        <CS>if</CS><AT>the</AT><NN1>system</NN1>
        <VVZ>displays</VVZ><AT1>an</AT1><JJ>alert</JJ><II>to</II>
        <AT>the</AT><NN1>user</NN1><IF>for</IF>
        <APPGE>its</APPGE><NN2>circumstances</NN2><SC>.</SC>
</Event>
...
```

```
                    ...
                <Event>
                        <CS>After</CS><VVN>completed</VVN><AT>the</AT>
                        <NN1>sending</NN1><SC>,</SC><AT>the</AT>
                        <NN1>system</NN1><VVZ>disconnects</VVZ><AT>the</AT>
                        <NN1>connection</NN1><SC>.</SC>
                </Event>
                <Event>
                        <AT>The</AT><NN1>phone</NN1><VVZ>displays</VVZ>
                        <AT>the</AT><NN1>status</NN1><IO>of</IO>
                        <VVG>sending</VVG><CC>and</CC><VVZ>keeps</VVZ>
                        <AT1>a</AT1><NN1>log</NN1><IO>of</IO>
                        <VVG>sending</VVG><SC>.</SC>
                </Event>
        </Flow_of_events>
        <Exceptional_events/>
        <Superordinate_use_case/>
        <Subordinate_use_case/>
</Use_Case>
```

Figure D- 1: Use case *sending a message*

```
<Use_Case UseCaseID="UC2" System="Mobile Phone" Product_Member="PM1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Z:\web\XTraQue\Use_case.xsd">
        <Title>
                <VVG>Making</VVG>
                <AT0>a</AT0>
                <NN1>call</NN1>
        </Title>
        <Description>
                <AT0>The</AT0><NN1>phone</NN1><VBZ>is</VBZ>
                <AJ0>able</AJ0><TO0>to</TO0>    <VVI>make</VVI>
                <AT0>a</AT0><NN1>call</NN1><SC>.</SC><AT0>The</AT0>
                <NN1>user</NN1><VM0>can</VM0><VVI>select</VVI>
                <AT0>a</AT0><AJ0>calling</AJ0><NN1>phone</NN1>
                <NN1>number</NN1><PRP>from</PRP><AT0>a</AT0>
                <NN1>list</NN1><PRF>of</PRF><NN1>phone</NN1>
                <NN2>numbers</NN2><DTQ>which</DTQ><VBB>are</VBB>
                <VVN>restored</VVN><PRP>in</PRP><AT0>the</AT0>
                <NN0>data</NN0><NN1>collection</NN1><CJC>or</CJC>
                <VVB>enter</VVB><AT0>the</AT0><NN1>number</NN1>
                <PRP>via</PRP><NN1>keypad</NN1><SC>.</SC><CJS>After</CJS>
                <AT0>the</AT0><NN1>user</NN1><VVZ>confirms</VVZ>
                <AT0>a</AT0><NN1>calling</NN1><AT0>the</AT0>
                <NN1>phone</NN1><VVZ>establishes</VVZ><AT0>the</AT0>
                <NN1>line</NN1><NN1>connection</NN1><TO0>to</TO0>
                <VVI>create</VVI><AT0>a</AT0><NN1>call</NN1>
                <SC>.</SC><CJS>If</CJS><RR>properly</RR><VDN>done</VDN>
                <AT0>the</AT0><NN1>phone</NN1><VVZ>dials</VVZ>
                <PRP>for</PRP><AT0>a</AT0><NN1>response</NN1>
                <PRP>from</PRP><AT0>the</AT0><NN1>receiver</NN1>
                <SC>.</SC><AV0>Otherwise</AV0><AT0>the</AT0>
                <NN1>phone</NN1><VVZ>informs</VVZ><AT0>the</AT0>
                <NN1>user</NN1><AT0>a</AT0><NN1>problem</NN1>
                <PRP>on</PRP><AT0>the</AT0><NN1>connection</NN1>
                <PRF>of</PRF>    <VVG>dialling</VVG><SC>.</SC>
                <PRP>In</PRP>    <AT0>the</AT0><NN1>case</NN1>
                <CJT>that</CJT><AT0>the</AT0><NN1>destination</NN1>
                <NN1>phone</NN1><VBZ>is</VBZ><VVN>engaged</VVN>
                <CJC>or</CJC><XX0>not</XX0><AJ0>able</AJ0>
                <TO0>to</TO0>    <VVI>reach</VVI><AT0>the</AT0>
                <NN1>signal</NN1><AT0>the</AT0><NN1>phone</NN1>
                <NN2>responses</NN2><AT0>a</AT0><NN1>voice</NN1>
                <NN1>message</NN1><PRP>to</PRP><AT0>the</AT0>
                <NN1>user</NN1><PRP>for</PRP><DPS>its</DPS>
                <NN2>circumstances</NN2><SC>.</SC>
        </Description>
        <Level>Primary task</Level>
        <Preconditions>
                <AT>The</AT><NN1>user</NN1><VHZ>has</VHZ>
                <VVN>selected</VVN><NN1>function</NN1><IO>of</IO>
                <VVG>making</VVG><AT1>a</AT1><NN1>call</NN1>
                <II>from</II><AT>the</AT><JJ>main</JJ><NN1>menu</NN1>
                <SC>.</SC></Preconditions> ...
        <Postconditions>
                <AT>The</AT><NN1>phone</NN1><VBZ>is</VBZ><JJ>ready</JJ>
                <IF>for</IF><MD>next</MD><NN2>actions</NN2><SC>.</SC>
        </Postconditions>
        <Primary_actor>The user</Primary_actor> ...
```

```
...
<Secondary_actors>-</Secondary_actors>
<Flow_of_events>
        <Trigger/>
        <Event> <AT>The</AT>    <NN1>phone</NN1><VBZ>is</VBZ>
                <JJ>ready</JJ><TO>to</TO><VVI>make</VVI>
                <AT1>a</AT1><NN1>call</NN1><SC>.</SC>
        </Event>
        <Event>
                <AT>The</AT><NN1>user</NN1><VVZ>selects</VVZ>
                <AT1>a</AT1><NN1>phone</NN1><NN1>number</NN1>
                <II>from</II><AT1>a</AT1><NN1>list</NN1>
                <IO>of</IO><NN2>contacts</NN2><CC>or</CC>
                <VVZ>enters</VVZ><AT1>a</AT1><NN1>phone</NN1>
                <NN1>number</NN1><II>via</II><NN1>keypad</NN1>
                <SC>.</SC>
        </Event>
        <Event>
                <AT>The</AT>    <NN1>user</NN1><VVZ>confirms</VVZ>
                <VVG>making</VVG><AT1>a</AT1><NN1>call</NN1>
                <SC>.</SC>
        </Event>
        <Event>
                <AT>The</AT><NN1>system</NN1><VVZ>establishes</VVZ>
                <AT>the</AT><NN1>line</NN1><NN1>connection</NN1>
                <SC>.</SC>
        </Event>
        <Event>
                <CS>If</CS><AT>the</AT><NN1>connection</NN1>
                <VBZ>is</VBZ><RR>properly</RR><VVN>set</VVN>
                <SC>,</SC><AT>the</AT><NN1>phone</NN1>
                <VVZ>dials</VVZ><AT>the</AT><NN1>number</NN1>
                <II>to</II><AT>the</AT><NN1>destination</NN1>
                <SC>.</SC><RR>Otherwise</RR><AT>the</AT>
                <NN1>phone</NN1><VVZ>informs</VVZ><AT>the</AT>
                <NN1>user</NN1><IF>for</IF><AT>the</AT>
                <JJ>existing</JJ><NN2>problems</NN2><SC>.</SC>
        </Event>
        <Event>
                <CS>If</CS><AT>the</AT><NN1>destination</NN1>
                <NN1>phone</NN1><VBZ>is</VBZ><JJ>engaging</JJ>
                <CC>or</CC><XX>not</XX><JK>able</JK>
                <TO>to</TO><VVI>reach</VVI><SC>,</SC>
                <AT>the</AT><NN1>phone</NN1><VVZ>informs</VVZ>
                <AT>the</AT><NN2>users</NN2><IF>for</IF>
                <APPGE>its</APPGE><NN2>circumstances</NN2><SC>.</SC>
        </Event>
        <Event>
                <AT>The</AT><NN1>user</NN1><VVZ>confirms</VVZ>
                <VVG>hanging</VVG><RP>up</RP><AT>the</AT>
                <NN1>call</NN1><SC>.</SC>
        </Event>
        <Event>
                <AT>The</AT><NN1>phone</NN1> <VVZ>disconnects</VVZ>
                <AT>the</AT><NN1>connection</NN1><SC>.</SC>
        </Event>
        ...
```

```
                ...
                <Event>
                        <AT>The</AT><NN1>phone</NN1><VVZ>shows</VVZ>
                        <NN1>usage</NN1><IO>of</IO><VVG>making</VVG>
                        <AT1>a</AT1><NN1>call</NN1><II>to</II>
                        <AT>the</AT><NN1>user</NN1><SC>.</SC>
                </Event>
                <Event>
                        <AT>The</AT><NN1>phone</NN1><VVZ>keeps</VVZ>
                        <AT1>a</AT1><NN1>log</NN1><NN1>file</NN1>
                        <IO>of</IO><VVG>making</VVG><AT1>a</AT1>
                        <NN1>call</NN1><II>at</II><AT>the</AT>
                        <NN1>moment</NN1><II>in</II><AT>the</AT>
                        <NN>data</NN><NN1>storage</NN1><SC>.</SC>
                </Event>
        </Flow_of_events>
        <Exceptional_events/>
        <Superordinate_use_case/>
        <Subordinate_use_case/>
</Use_Case>
```

Figure D- 2: Use case *making a call*

```
<Use_Case UseCaseID="UC3" System="Mobile Phone" Product_Member="PM1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Z:\web\XTraQue\Use_case.xsd">
        <Title>
                <VVG>Taking</VVG>
                <AT0>a</AT0>
                <NN1>picture</NN1>
        </Title>
        <Description>
                <AT0>The</AT0><NN1>phone</NN1><VBZ>is</VBZ>
                <VVN>integrated</VVN><PRP>with</PRP><AT0>a</AT0>
                <AJ0>digital</AJ0><NN1>camera</NN1><SC>.</SC>
                <PNP>It</PNP><VVZ>enables</VVZ><AT0>a</AT0>
                <NN1>user</NN1><VVG>taking</VVG><CJC>and</CJC>
                <VVG>restoring</VVG><AT0>a</AT0><NN1>picture</NN1>
                <PRP>in</PRP><AT0>the</AT0><NN1>phone</NN1>
                <SC>.</SC><AT0>The</AT0><NN1>photo</NN1>
                <NN1>file</NN1><VBZ>is</VBZ><NP0>JPG</NP0>
                <NN1>format</NN1><SC>.</SC><AT0>The</AT0>
                <NN1>photo</NN1><VBZ>is</VBZ><AV0>possibly</AV0>
                <VVN>taken</VVN><CJS>as</CJS> <CRD>one</CRD>
                <PRF>of</PRF><CRD>three</CRD><AJ0>optional</AJ0>
                <NN2>types</NN2><AV0>i.e.</AV0><NN1>general</NN1>
                <NN1>night</NN1><CJC>and</CJC><NN1>portrait</NN1>
                <DTQ>which</DTQ><VBB>are</VBB><AJ0>different</AJ0>
                <AJ0>sized</AJ0><SC>.</SC><AV0>Also</AV0><AT0>the</AT0>
                <NN2>photos</NN2><CJT>that</CJT><VBB>are</VBB>
                <VVN>kept</VVN><PRP>in</PRP><AT0>the</AT0><NN1>phone</NN1>
                <VM0>can</VM0><VBI>be</VBI><VVN>viewed</VVN>
                <CJC>and</CJC><VVN>deleted</VVN><AV0>afterwards</AV0>
                <SC>.</SC>
        </Description>
        <Level>Primary task</Level>
        <Preconditions>    <AT>The</AT>
                <NN1>user</NN1><VHZ>has</VHZ><VVN>selected</VVN>
                <NN1>function</NN1><IO>of</IO><VVG>taking</VVG>
                <AT1>a</AT1><NN1>photo</NN1><II>from</II><AT>the</AT>
                <JJ>main</JJ><NN1>menu</NN1><SC>.</SC></Preconditions>
        <Postconditions>
                <AT>The</AT><NN1>phone</NN1><VVD>took</VVD><AT1>a</AT1>
                <NN1>photo</NN1><CC>and</CC><VVD>kept</VVD><PPH1>it</PPH1>
                <II>as</II><AT1>a</AT1><JJ>JPG-formatted</JJ> <NN1>file</NN1>
                <II>in</II><APPGE>its</APPGE><JJ>temporary</JJ><NN1>memory</NN1>
                <NN1>storage</NN1><BCL>in</BCL><BCL>order</BCL><TO>to</TO>
                <VBI>be</VBI><VVN>restored</VVN><AT>the</AT><NN>data</NN>
                <NN1>collection</NN1><RRR>later</RRR><RP>on</RP><SC>.</SC>
                <RT>Then</RT><AT>The</AT><NN1>phone</NN1><VBZ>is</VBZ>
                <JJ>ready</JJ><IF>for</IF><VVG>capturing</VVG><MD>next</MD>
                <NN2>shots</NN2><SC>.</SC></Postconditions>
        <Primary_actor>The user</Primary_actor>
        <Secondary_actors>-</Secondary_actors>
        <Flow_of_events>
                <Trigger/>
                <Event><AT>The</AT><NN1>system</NN1><VVZ>shows</VVZ>
                <AT1>a</AT1><NN1>list</NN1><IO>of</IO><JJ>optional</JJ>
                <NN2>types</NN2><SC>.</SC><NN2>Types</NN2><IF>for</IF> ...
```

```
                ...
                <VVG>taking</VVG><AT1>a</AT1><NN1>photo</NN1><REX>i.e.</REX>
                <NN1>general</NN1><NNT1>night</NNT1><CC>and</CC>
                <NN1>portrait</NN1><SC>.</SC></Event>
        <Event>
                <AT>The</AT><NN1>user</NN1><VVZ>selects</VVZ>
                <MC1>one</MC1><IO>of</IO><JJ>optional</JJ><NN2>types</NN2>
                <SC>.</SC><NN2>Types</NN2><IO>of</IO><VVG>taking</VVG>
                <AT1>a</AT1><NN1>photo</NN1><SC>.</SC>
        </Event>
        <Event>
                <AT>The</AT><NN1>system</NN1><VVZ>shows</VVZ><AT>the</AT>
                <NN1>scenario</NN1><II>on</II><AT>the</AT><NN1>screen</NN1>
                <SC>.</SC>
        </Event>
        <Event>
                <AT>The</AT><NN1>user</NN1><NN2>clicks</NN2><AT>the</AT>
                <NN1>button</NN1><II>on</II><AT>the</AT><NN1>phone</NN1>
                <TO>to</TO><VVI>capture</VVI><AT1>a</AT1><NN1>shot</NN1>
                <SC>.</SC>
        </Event>
        <Event>
                <AT>The</AT><NN1>system</NN1><VVZ>displays</VVZ>
                <AT>the</AT><NN1>shot</NN1><VVN>done</VVN><II>at</II>
                <AT>the</AT><NN1>moment</NN1><SC>.</SC></Event>
        <Event>
                <AT>The</AT><NN1>system</NN1><VVZ>pops</VVZ>
                <RP>up</RP><AT1>a</AT1><NN1>request</NN1><IO>of</IO>
                <VVG>restoring</VVG><AT>the</AT><NN1>shot</NN1><II>as</II>
                <AT1>a</AT1><NN1>photo</NN1><II>in</II><AT>the</AT>
                <NN1>phone</NN1><SC>.</SC>
        </Event>
        <Event>
                <CS>If</CS><AT>the</AT><NN1>user</NN1><VVZ>wants</VVZ>
                <TO>to</TO><VVI>keep</VVI><AT>the</AT><NN1>shot</NN1>
                <SC>.</SC><AT>the</AT><NN1>system</NN1><VVZ>restores</VVZ>
                <AT>the</AT><NN1>photo</NN1><II>as</II><AT1>a</AT1>
                <NN1>file</NN1><II>in</II><AT>the</AT><NN>data</NN>
                <NN1>collection</NN1><SC>.</SC>
        </Event>
        <Event>
                <AT>The</AT><NN1>system</NN1><VVZ>shows</VVZ><AT>the</AT>
                <NN1>scenario</NN1><II>on</II><AT>the</AT><NN1>screen</NN1>
                <TO>to</TO><VBI>be</VBI><JJ>ready</JJ><IF>for</IF><MD>next</MD>
                <NN2>snapshots</NN2><SC>.</SC>
        </Event>
        </Flow_of_events>
        <Exceptional_events/>
        <Superordinate_use_case></Superordinate_use_case>
        <Subordinate_use_case></Subordinate_use_case>
</Use_Case>
```

Figure D- 3: Use case *taking a picture*

```
<Use_Case UseCaseID="UC4" System="Mobile Phone"  Product_Member="PM1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="Z:\web\XTraQue\Use_case.xsd">
        <Title>
                <NN1>Sending</NN1>
                <VVZ>emails</VVZ>
        </Title>
        <Description>
                <AT0>The</AT0> <NN1>user</NN1><VBZ>is</VBZ><AJ0>able</AJ0>
                <TO0>to</TO0><VVI>send</VVI><NN2>emails</NN2>
                <PRP>with</PRP><NN1>attachment</NN1><PRP>via</PRP>
                <NN1>network</NN1><NN2>protocols</NN2><AV0>e.g.</AV0>
                <NP0>SMTP</NP0><UNC>POP3</UNC><UNC>IMAP4</UNC>
                <SC>.</SC><AT0>The</AT0><NN1>user</NN1><VM0>can</VM0>
                <VVI>specify</VVI><AT0>the</AT0><NN1>address(s)</NN1>
                <PRF>of</PRF><NN2>recipient(s)</NN2><PRP>by</PRP>
                <VVG>selecting</VVG><PRP>from</PRP><AT0>a</AT0>
                <NN1>list</NN1><PRF>of</PRF><NN2>contacts</NN2>
                <DTQ>which</DTQ><VBB>are</VBB><VVN>restored</VVN>
                <PRP>in</PRP><AT0>the</AT0><NN0>data</NN0><NN1>collection</NN1>
                <PRF>of</PRF><AT0>the</AT0><NN1>phone</NN1><CJC>or</CJC>
                <VVB>enter</VVB><PRP>via</PRP><NN1>keypad</NN1>
                <SC>.</SC><AT0>The</AT0><NN1>user</NN1><VM0>can</VM0>
                <VVI>send</VVI><NN2>emails</NN2><PRP>to</PRP> <AJ0>multiple</AJ0>
                <NN2>receivers</NN2><PRP>in</PRP><CRD>one</CRD>
                <NN1>time</NN1><SC>.</SC><VVG>Sending</VVG><VBZ>is</VBZ>
                <VVN>limited</VVN><PRP>under</PRP><AT0>the</AT0>
                <AJ0>maximum</AJ0><NN1>size</NN1><SC>.</SC><AT0>The</AT0>
                <NN1>user</NN1><VM0>can</VM0><VVI>attach</VVI><DT0>some</DT0>
                <NN2>files</NN2><PRF>of</PRF><NN2>notes</NN2><NN0>txt</NN0>
                <NN2>photos</NN2><NN0>jpg</NN0><CJC>and</CJC><NN2>images</NN2>
                <NN0>jpg</NN0><PRP>in</PRP><VVG>sending</VVG><NN2>emails</NN2>
                <SC>.</SC><AT0>The</AT0><NN1>phone</NN1><VVZ>keeps</VVZ>
                <AT0>a</AT0><NN1>log</NN1><NN1>file</NN1><PRF>of</PRF>
                <VVG>sending</VVG><AT0>an</AT0><NN1>email</NN1>
                <PRP>in</PRP><AT0>the</AT0><NN1>storage</NN1><AT0>the</AT0>
                <NN1>user</NN1><VM0>can</VM0><VVI>view</VVI>
                <CJC>and</CJC><VVI>delete</VVI><AT0>the</AT0><NN1>log</NN1>
                <NN2>files</NN2><AV0>later</AV0><AVP>on</AVP><SC>.</SC>
        </Description>
        <Level>Primary task</Level>
        <Preconditions>
                <AT>the</AT><NN1>user</NN1><VHZ>has</VHZ><RR>already</RR>
                <VVN>selected</VVN><AT1>a</AT1><NN1>function</NN1>
                <IO>of</IO><VVG>sending</VVG><AT1>an</AT1><NN1>email</NN1>
                <II>from</II><AT>the</AT><JJ>main</JJ><NN1>menu</NN1>
        </Preconditions>
        <Postconditions>
                <AT>the</AT><VHZ>phone</VHZ><VHZ>has</VHZ><VVN>sent</VVN>
                <NN1>email</NN1><II>to</II><AT>the</AT><NN2>receiver(s)</NN2>
                <CC>and</CC><VBN>been</VBN><VVG>showing</VVG>
                <AT>the</AT><NN1>response</NN1><II>to</II><AT>the</AT>
                <NN1>user</NN1>
        </Postconditions>
        <Primary_actor>The user</Primary_actor>
        <Secondary_actors>-</Secondary_actors>
        <Flow_of_events>
                <Trigger/>
                <Event>  <AT>the</AT><NN1>system</NN1><VVZ>shows</VVZ>
                        <AT1>an</AT1><NN1>editor</NN1><VVN>composed</VVN>
                        <IO>of</IO><AT1>a</AT1><NN1>text</NN1><NN1>box</NN1>
                        <IF>for</IF><VVG>specifying</VVG><AT>the</AT>
                        <NN1>email</NN1><NN1>address(s)</NN1> ...
```

```
...
<IO>of</IO><NN2>receiver(s)</NN2><CC>and</CC>
<AT1>a</AT1><JJ>blank</JJ><NN1>note</NN1>
<IF>for</IF><VVG>writing</VVG><AT1>a</AT1>
<NN1>message</NN1>
</Event>
<Event>
<AT>The</AT><NN1>user</NN1><VVZ>inserts</VVZ>
<AT1>an</AT1><NN1>email</NN1><NN1>address(s)</NN1>
<IO>of</IO><NN2>receiver(s)</NN2><II>by</II>
<VVG>selecting</VVG><II>from</II><AT1>a</AT1>
<NN1>list</NN1><IO>of</IO><NN2>contacts</NN2>
<VVG>restoring</VVG><II>in</II><AT>the</AT>
<NN>data</NN><NN1>collection</NN1><IO>of</IO>
<AT>the</AT><NN1>phone</NN1><CC>or</CC>
<VVG>entering</VVG><II>via</II><NN1>keypad</NN1>
<VV0>Note</VV0><CST>that</CST><AT>the</AT>
<NN1>user</NN1><VM>can</VM><VVI>send</VVI>
<AT>the</AT><NN1>email</NN1><II>to</II>
<JJ>multiple</JJ><NN2>receivers</NN2><II>by</II>
<VVG>separating</VVG><AT>the</AT><NN1>email</NN1>
<NN2>addresses</NN2><IW>with</IW><FU>*</FU><SC>.</SC>
</Event>
<Event>
<AT>The</AT><NN1>user</NN1><VM>can</VM>
<VVI>enter</VVI><AT>the</AT><NN1>message</NN1>
</Event>
<Event>
<AT>The</AT><NN1>user</NN1><VM>may</VM>
<VVI>attach</VVI><AT>the</AT><NN1>email</NN1>
<IW>with</IW><DD>any</DD><NN2>files</NN2>
<IO>of</IO><NN2>notes</NN2><NNU>txt</NNU>
<NN2>photos</NN2><NNU>jpg</NNU><CC>and</CC>
<NN2>images</NN2><NNU>jpg</NNU><CST>that</CST>
<VBR>are</VBR><JJ>available</JJ><II>in</II>
<AT>the</AT><NN1>phone</NN1><RR>Otherwise</RR>
<DD1>this</DD1><VBZ>is</VBZ><VVN>limited</VVN>
<II>under</II><AT>the</AT><JJ>maximum</JJ>
<NN1>size</NN1><IO>of</IO><VVG>sending</VVG>
<NN2>emails</NN2><VV0>Note</VV0><CST>that</CST>
<AT>the</AT><NN1>event</NN1><IO>of</IO>
<MC>2</MC><MC>3</MC><CC>and</CC><MC>4</MC>
<VBR>are</VBR><XX>not</XX><JJ>sequential</JJ>
<NN2>processes</NN2></Event>
<Event>
<AT>The</AT><NN1>user</NN1><VVZ>confirms</VVZ>
<VVG>sending</VVG><AT>the</AT><NN1>email</NN1>
</Event>
<Event>
<AT>The</AT><NN1>phone</NN1><VVZ>establishes</VVZ>
<AT>the</AT><NN1>connection</NN1><IF>for</IF>
<NN1>email</NN1><VVG>sending.</VVG>
</Event>
<Event>
<CS>if</CS><AT>the</AT><NN1>connection</NN1>
<VBZ>is</VBZ><RR>properly</RR> <VVN>set</VVN>
<AT>the</AT><NN1>phone</NN1><VVZ>sends</VVZ>
<AT>the</AT><NN1>email</NN1><II>via</II>
<AT>the</AT><NN1>network</NN1><NN2>protocols</NN2>
<RR>Otherwise</RR><CS>If</CS><AT>the</AT>
<NN1>phone</NN1><VVZ>informs</VVZ><AT>the</AT>
<NN2>users</NN2><IF>for</IF><APPGE>its</APPGE>
<NN2>circumstances</NN2>
```

```
                    ...
                </Event>
                <Event>
                        <CS>After</CS><VVN>completed</VVN><AT>the</AT>
                        <VVG>sending</VVG><AT>the</AT><NN1>email</NN1>
                        <AT>the</AT><NN1>phone</NN1><VVZ>disconnects</VVZ>
                        <AT>the</AT><NN1>connection</NN1>
                </Event>
                <Event>
                        <AT>The</AT><NN1>phone</NN1><VVZ>shows</VVZ>
                        <AT>the</AT><NN1>status</NN1><IO>of</IO>
                        <VVG>sending</VVG><AT>the</AT><NN1>email</NN1>
                        <RT>then</RT><VVZ>keeps</VVZ><AT1>a</AT1>
                        <NN1>log</NN1><NN1>file</NN1><IO>of</IO>
                        <VVG>sending</VVG><AT>the</AT><NN1>email</NN1>
                        <II>at</II><AT>the</AT><NN1>moment</NN1><II>in</II>
                        <AT>the</AT><NN>data</NN><NN1>storage</NN1>
                </Event>
        </Flow_of_events>
        <Exceptional_events/>
        <Superordinate_use_case></Superordinate_use_case>
        <Subordinate_use_case></Subordinate_use_case>
</Use_Case>
```

Figure D- 4: Use case *sending emails*

D.2. Class Diagram – PM1

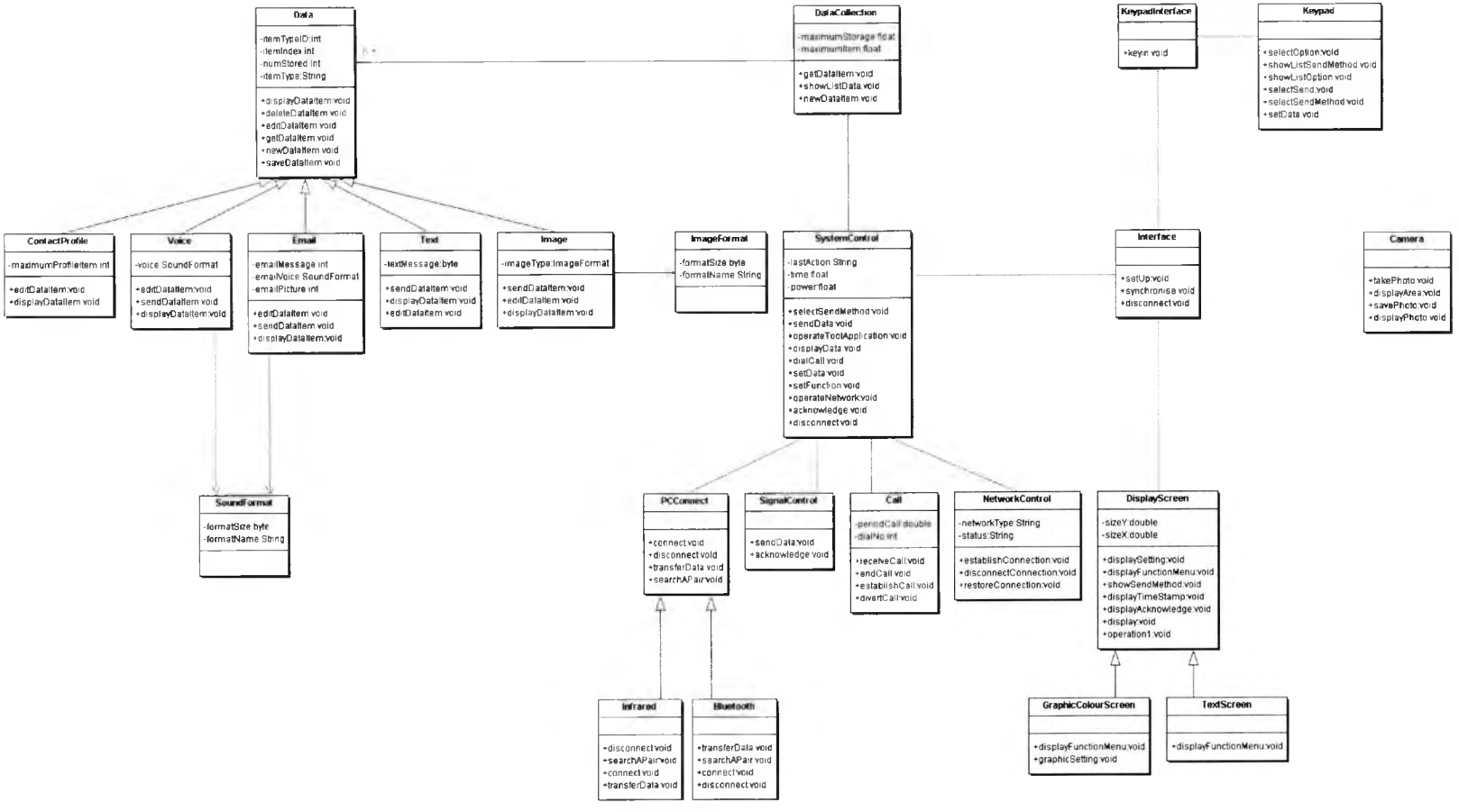


Figure D- 5: A class diagram of product member PM1

D.3. Sequence Diagram – PM1

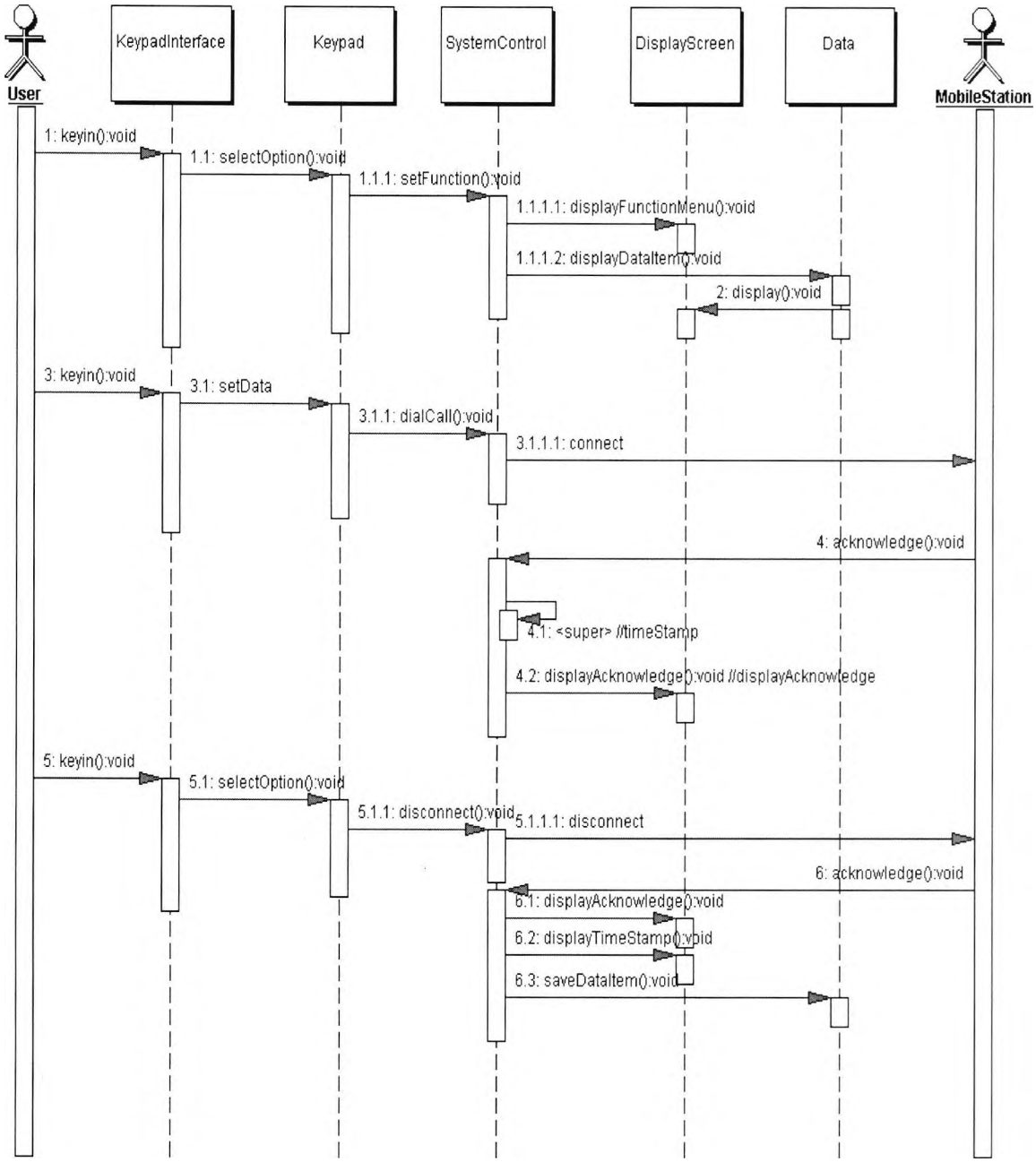


Figure D- 6: A sequence diagram *Making a call*

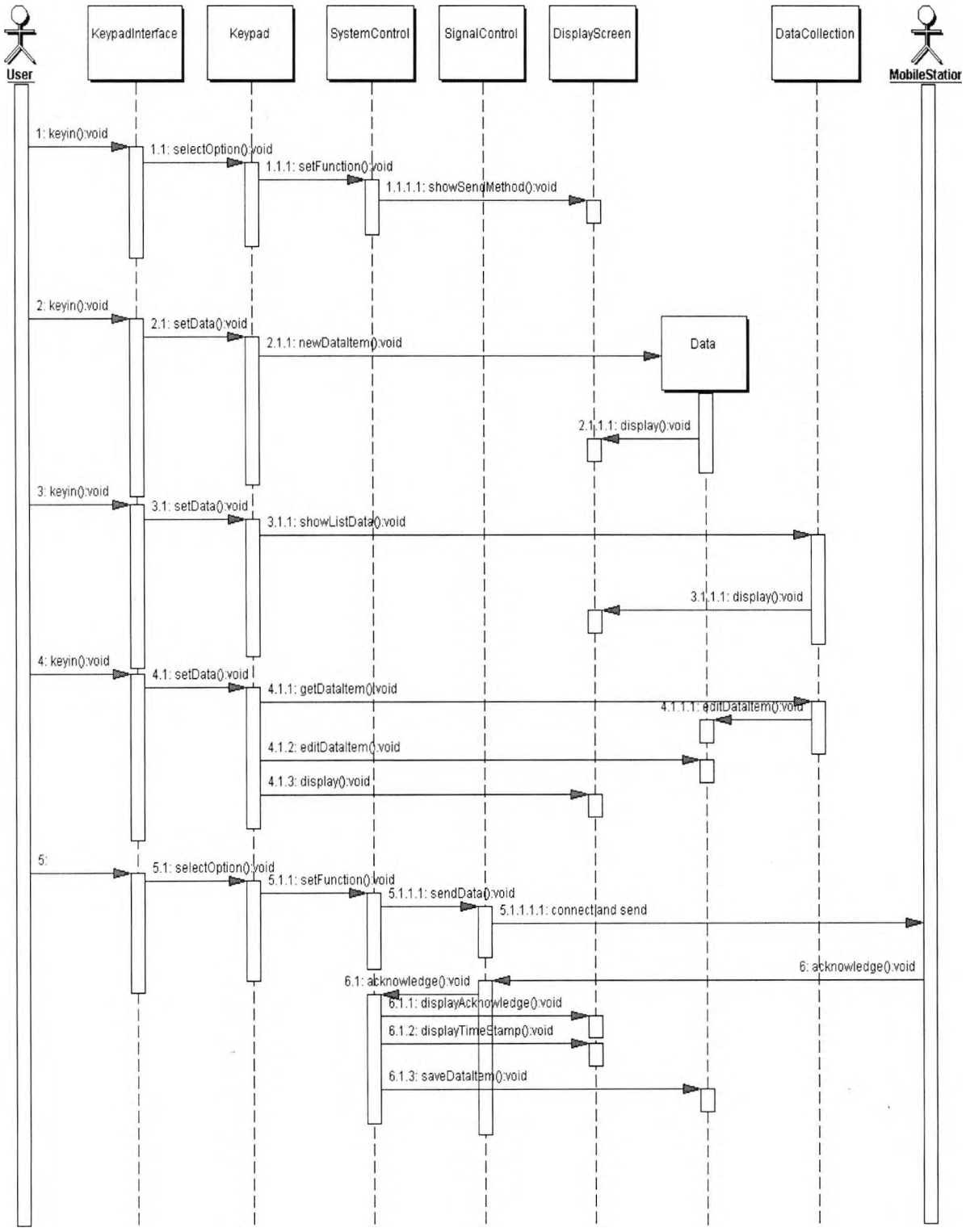


Figure D- 7: A sequence diagram *Sending data*

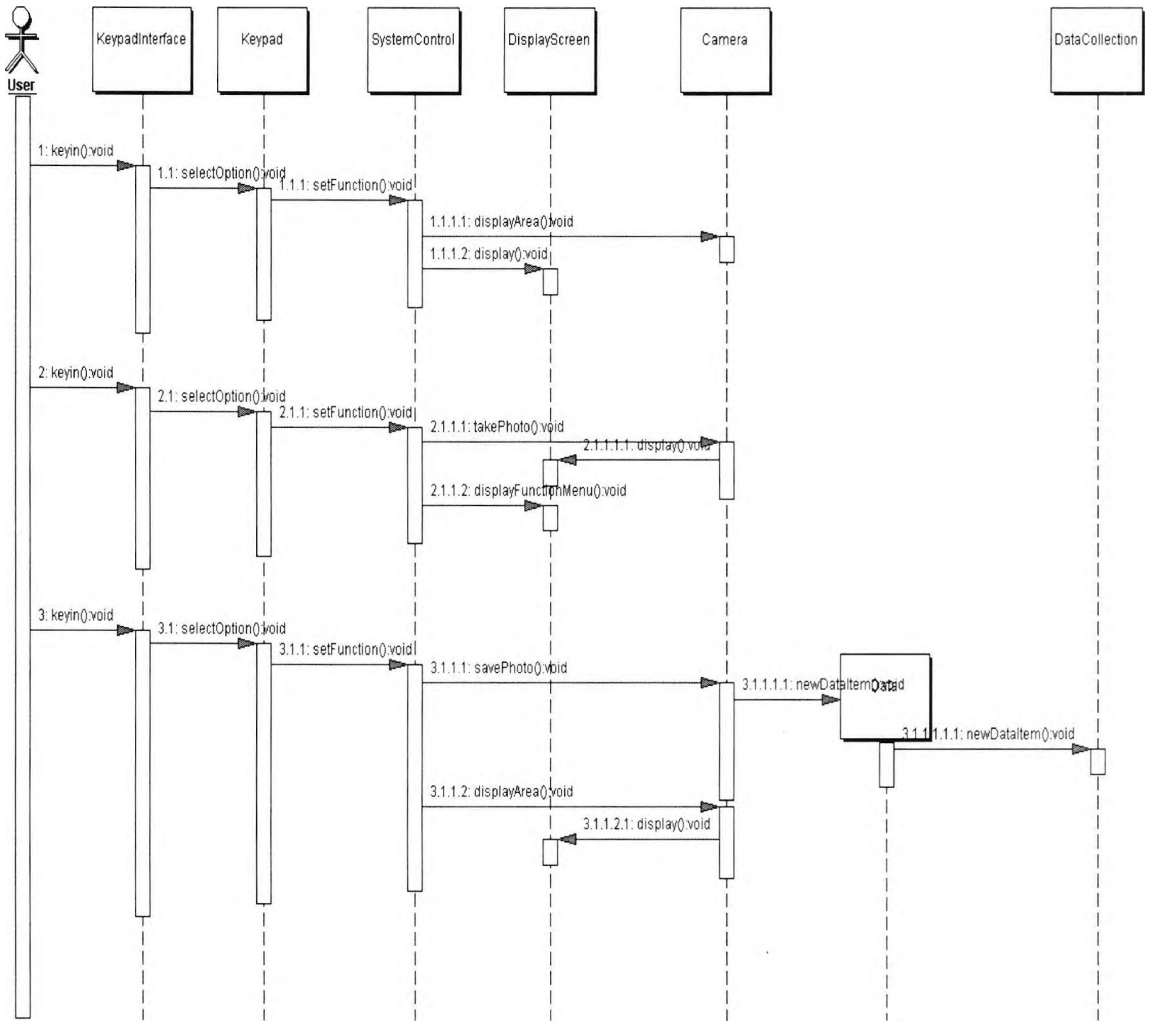


Figure D- 8: A sequence diagram *Taking a photo*

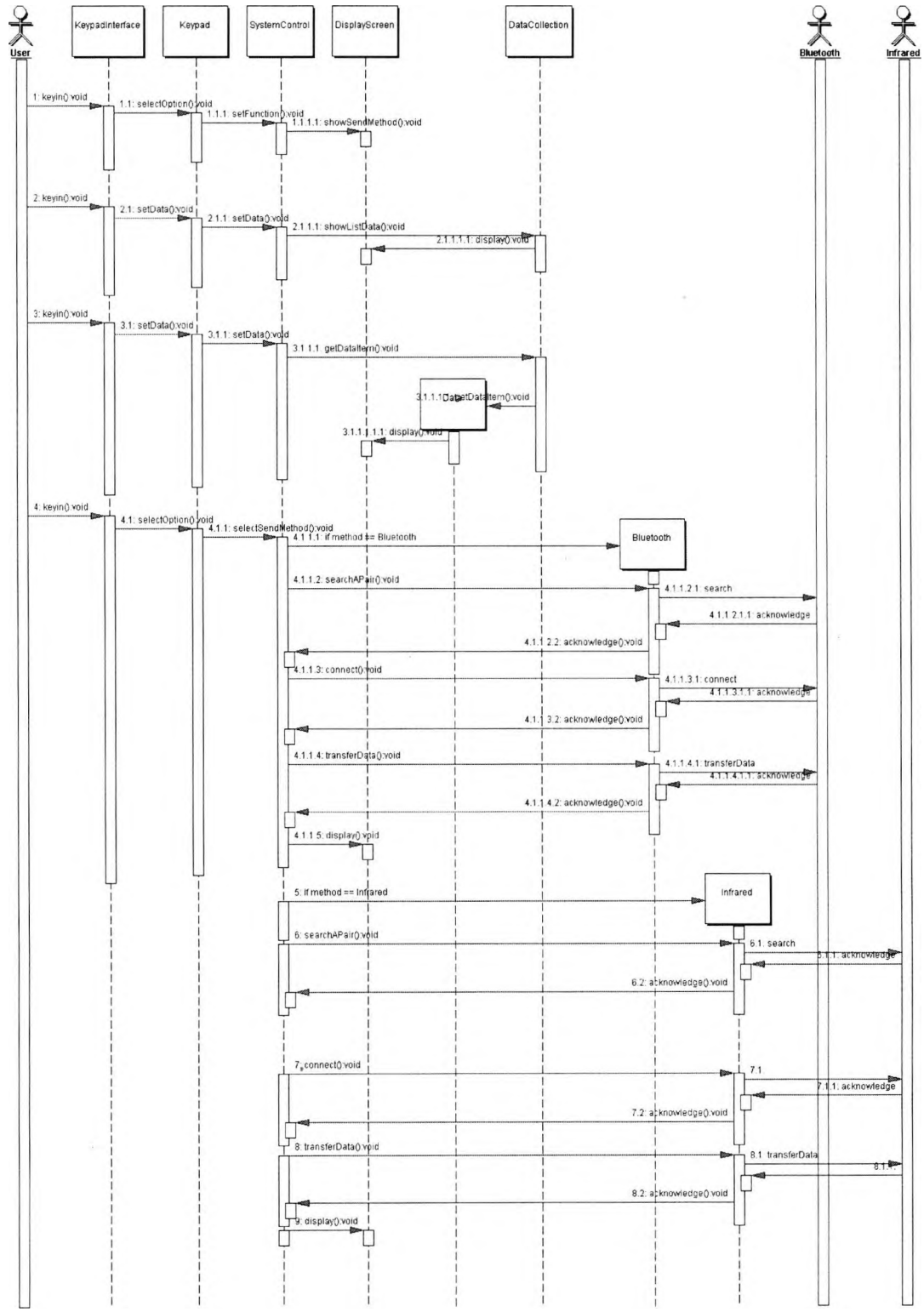


Figure D- 9: A sequence diagram *Transfering data*

D.4. Statechart Diagram – PM1

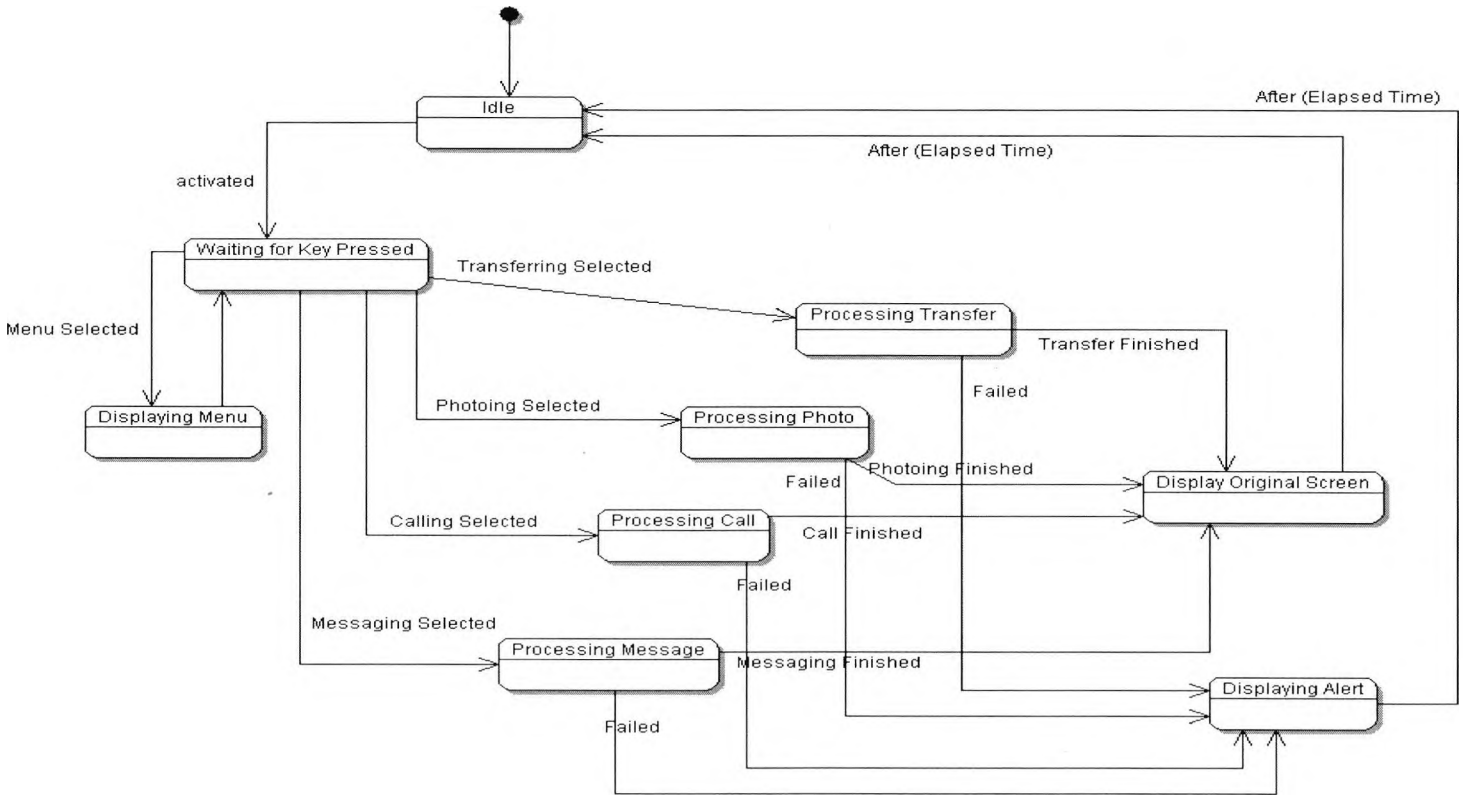


Figure D- 10: A statechart diagram of product member PM1

BIBLIOGRAPHY

3SL. CRADLE. from http://3sl.co.uk

Alexander, I. 2003. SemiAutomatic Tracing of Requirement Versions to Use Cases – Experience and Challenges. *the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003)*, Montreal, Canada.

America, P., H. Obbink., J. Muller, and R. Van Ommering. 2000. COPA: A Component-Oriented Platform Architecting Method for Families of Software Intensive Electronic Products. *Tutorial in: The First Conference on Software Product Line Engineering (SPLC1)*, Denver, Colorado.

Anderson, K. M., S. A. Sherba, and W. V. Lepthien. 2002. Towards Large-Scale Information Integration. Pages 524-535. *the 24th International Conference on Software Engineering*, Orlando, FL, USA.

Antoniol, G., G. Canfora, G. Casazza, and A. De Lucia. 2000. Information Retrieval Models for Recovering Traceability Pages 40-51. *IEEE International Conference on Software Maintenance (ICSM'00)*, San Jose, CA.

Antoniol, G., G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo. 2002. Recovering Traceability Links between Code and Documentation. *IEEE Transactions on Software Engineering* 28: 970-983.

Arango, G., and R. Prieto-Diaz. 1991. Domain Analysis Concepts and Reseach Directions. *Domain Analysis and Software Systems Modelings*: 9-31.

Ardis, M. A., and D. M. Weiss. 1997. Defining Families: The Commonality Analysis. Pages 649-650. *the 19th International Conference on Software Engineering*. ACM Press New York, NY, USA, Boston, Massachusetts, United States.

ArgoUML. from http://argouml.tigris.org/project.html.

ASADAL. from selab.postech.ac.kr/form/.

Atkinson, C., J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wust, and J. Zettel. 2002. *Component-based Product Line Engineering with UML*. Addison-Wesley.

Atkinson, C., J. Bayer, and D. Muthig. 2000. Component-based product line development: The KobrA approach. Pages 289-310. *the 1st Software Product Line Conference, SPLC*. Kluwer, Denver, Colorado, USA.

Bailin, S., et al. 1990. KAPTUR: Knowledge Acquisition for Preservation of Tradeoffs and Underlying Rationale. Pages 95-104. *Proceedings of the 5th Annual Knowledge-Based Software Assistant Conference*, Liverpool, NY,Rome, NY: Rome Air Development Center.

Bass, L., P. Clements, and R. Kazman. 2003. *Software Architecture in Practice.* Addison-Wesley Professional.

Bastarrica, M. C., N. Hitschfeld-Kahler, and P. Rossel. 2006. Product Line Architecture for a Family of Meshing Tools. Pages 403 - 406. *9th International Conference on Software Reuse (ICSR)*, Turin, Italia.

Batory, D., R. Cardone, and Y. Smaragdakis. 2000. Object-Oriented Frameworks and Product-Lines. Pages 227-247. *the 1st Software Product-Line Conference* Denver, Colorado, United States.

Bayer, J., O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. 1999. PuLSE: A methodology to develop software product lines. Pages 122-131. *the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99)*, Los Angeles, CA, USA.

Bayer, J., and T. Widen. 2001. Introducing Traceability to Product Lines. Pages 409-416. *the 4th International Workshop on Software Product-Family Engineering (PFE 2001)*. Springer-Verlag, Bilbao, Spain.

—. 2002. Introducing Traceability to Product Lines, Software Product-Family Engineering. Pages 409-416. *the 4th International Workshop, PFE 2001.* Springer Verlag, Bilbao, Spain.

Berg, K., and J. Bishop. 2005. Tracing Software Product Line Variability - From Problem to Solution Space. Pages 111-120. *SAICSIT 2005.*

Boehm, B. 2000. *Software Cost Estimation with Cocomo II.* Upper Saddle River, NJ: Prentice Hall.

Boehm, B., A. W. Brown, R. Madachy, and Y. Yang. 2004. A Software Product Line Life Cycle Cost Estimation Model. Pages 156-164. *Proceedings of the 2004 International Symposium on Empirical Software Engineering.* Los Alamitos, CA: IEEE Computer Society, Redondo Beach, CA.

Borland. Borland Together Control Center 6.2.

Bosch, J. 1998. Product-Line Architectures in Industry: A Case Study. Pages 544 - 554. *the 21st International Conference on Software Engineering.* IEEE Computer Society Press, Los Angeles, California, United States.

—. 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach.* Addison Wesley.

—. 2001. Software Product Lines: Organizational Alternatives. *the 23rd International Conference on Software Engineering.*

Bosch, J., and M. Hogstrom. 2000. Product Instantiation in Software Product Lines: A Case Study. Pages 147-162. *the Second International Symposium on Generative and Component-Based Software Engineering (GCSE 2000).* Springer-Verlag London, UK.

CAFE. 2003. from http://www.esi.es/en/projects/cafe/cafe.html.

CaliberRM. from http://www.starbase.com.

Campbell, G. H., Jr., S. R. Faulk, and D. M. Weiss. 1990. Introduction To Synthesis, INTRO_SYNTHESIS_PROCESS-90019-N. *Software Productivity Consortium*, Herndon, VA, USA.

Clauss, M. 2001. Modeling variability with UML. *GCSE 2001 - Young Researchers Workshop.*

CLAWS. from https://www.comp.lancs.ac.uk/ucrel/claws.

Cleland-Huang, J., C. K. Chang, and Y. Ge. 2002a. Supporting Event Based Traceability through High-Level Recognition of Change Events. *the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*, Oxford, England.

Cleland-Huang, J., C. K. Chang, G. Sethi, K. Javvaji, H. Hu, and J. Xia. 2002b. Automating Speculative Queries through Event-based Requirements Traceability. *International Requirements Engineering Conference*, Essen, Germany.

Cleland-Huang, J., R. Settimi, O. BenKhadra, E. Berezhanskaya, and S. Christina. 2005a. Goal-centric traceability for managing non-functional requirements. Pages 362 - 371 *the 27th international conference on Software engineering.* ACM Press New York, NY, USA, St. Louis, MO, USA.

Cleland-Huang, J., R. Settimi, C. Duan, and X. Zou. 2005b. Utilizing Supporting Evidence to Improve Dynamic Requirements Traceability. *13th IEEE International Conference on Requirements Engineering (RE'05).*

Cleland-Huang, J., G. Zemont, and W. Lukasik. 2004. A Heterogeneous Solution for Improving the Return on Investment of Requirements Traceability *12th IEEE International Requirements Engineering Conference (RE'04).*

Clements, P., and L. Northrop. 2002. *Software Product Lines: Practices and Patterns.* Addison-Wesley, Boston, MA.

—. 2004. A Framework for Software Product Lines Practice. http://www.sei.cmu.edu/productlines/framework.html

COBRA. from http://www.omg.org/.

Cockburn, A. 1997. Structuring Use-Cases With Goals. *Journal of Object-Oriented Programming* Sep/Oct: 35-40.

—. 2000. *Writing Effective Use Cases.* Addison-Wesley, Boston

COM. from http://www.microsoft.com/com/default.mspx.

CORE. from www.vtcorp.com.

Coriat, M., J. Jourdan, and F. Boisbourdin. 2000. The SPLIT Method. Pages 147-166. *the First Software Product Lines Conference (SPLC1)*, Denver, Colorado, USA.

Dean, J. G., A. (Eds.) 2002. COTS-Based Software Systems. *First International Conference, ICCBSS.* Springer-Verlag, Orlando, FL, USA.

Department_of_Defense. 1996. Software Reuse Executive Primer. Falls Church, VA.

Dhar, V., and M. Jarke. 1988. Dependency Directed Reasoning and Learning in Systems Maintenance Support. *IEEE Transactions in Software Engineering* 14: 211-227.

Dick, J. 1999. Rich Traceability. http://www.telelogic.com/industries/telecoms/papers.cfm

Dömges, R., and K. Pohl. 1998. Adapting Traceability Environments to Project Specific Needs. *Communications of the ACM* 41: 54-62.

DOORS. from www.telelogic.com/products/doors.

Dorfman, M., and R. F. Flynn. 1984. Arts - An Automated Requirements Traceability System. *The Journal of Systems and Software* 4: 63-74.

Egyed, A. 2001. A Scenario-Driven Approach to Traceability. *the 23rd International Conference on Software Engineering (ICSE 2001)*, Toronto, Canada.

—. 2002. Reasoning about Trace Dependencies in a Multi-Dimensional Space. *the 1st International Workshop on Traceability, co-located with ASE 2002*, Edinburgh, Scotland, UK.

—. 2003. A Scenario-Driven Approach to Trace Dependency Analysis. *IEEE Transactions on Software Engineering* 9.

Egyed, A., and P. Grunbacher. 2002. Automatic Requirements Traceability: Beyond the Record and Replay paradigm. *the 17th IEEE International Conference on Automated Software Engineering (ASE)*, Edinburgh, UK.

Egyed, A., and P. Grunbacher. 2003. Towards Understanding Implications of Trace Dependencies among Quality Requirements. *the 2nd International Workshop on Traceability in Emerging Form Software Engineering (TEFSE'03).*

ESAPS. from http://www.esi.es/en/Projects/esaps/esaps.html.

Fairley, R. E., and R. H. Thayer. 1997. The Concept of Operations: The Bridge from Oper-ational Requirements to Technical Specifications in M. Dorfman and T. R. J, eds. *Software Engineering.* IEEEComp. Press, Los Alamitos, CA.

Faloutsos, C., and D. W. Oard. 1995. A Survey of Information Retrieval and Filtering Methods. Dept. of Computer Science, Univ. of Maryland.

Fantechi, A., S. Gnesi, G. Lami, and E. Nesti. 2004. A Methodology for the Derivation and Verification of Use Casees for Product Lines. Pages 255-264. *the 3rd International Conference, SPLC 2004.* Springer Verlag, Boston, MA, USA.

Finkelstein, A. 1991. Tracing Back from Requirements. *IEE Colloquium on Tools & Techniques for Maintaining Traceability During Design.*

Finkelstein, A., and H. Fuks. 1989. Multi-Party Specification Pages 185-199. *5th International Workshop on Software Specification & Design.*

Finkelstein, A., J. Kramer, and M. Goedicke. 1990. ViewPoint Oriented Software Development. Pages 337-351. *3rd International Workshop Software Engineering & its Applications.* Cigref EC2 V1.

Finkelstein, W., and J. A. R. Guertin. 1998. Integrated Logistics Support. *The Design Engineering Link, IFS Publications.* Springer Verlag.

Fiutem, R., and G. Antoniol. 1998. Identifying Design-Code Inconsistencies in Object-Oriented Software: a Case Study. Pages 94 *the International Conference on Software Maintenance table of contents (ICSM).*

GEARS. from http://www.biglever.com/

Gibson, P., B. Mermet, and D. Méry. 1997. Feature Interactions: A Mixed Semantic Model Approach in O'Regan and Flynn, eds. *1st Irish Workshop on Formal Methods (IWFM97)*, Dublin.

Gomaa, H. 1993. *Software Design Methods for Concurrent and Real-Time Systems.* Addison Wesley.

—. 2004. *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures.* Addison Wesley Professional.

Gomaa, H., R. Fairley, and L. Kerschberg. 1989. Towards an evolutionary domain life cycle model. *In Workshop on Domain Modeling for Software Engineering.*

Gomaa, H., and M. E. Shin. 2004. A Multiple-View Meta-modeling Approach for Variability Management in Software Product Lines. Pages 274-285 in J. Bosch and C. Krueger, eds. *8th International Conference (ICSR 2004).* Springer Verlag, Madrid, Spain.

Gotel, O., and A. Finkelstein. 1994. An Analysis of the Requirements Traceability Problem. Pages 94 -101. *the First International Conference on Requirements*, England.

—. 1995. Contribution Structure. Pages 100-107. *the Second IEEE International Symposium on Requirements Engineering (RE'95).* IEEE Computer Society Press, York.

Griss, M. L. 2000. Implementating Product-Line Features with Component Reuse. *the 6th International Conference on Software Reuse.* Springer-Verlag, Austria.

Griss, M. L., J. Favaro, and M. d. Alessandro. 1998. Integrating feature modeling with the RSEB. Pages 76-85 in P. Devanbu and J. Poulin, eds. *the 5th International Conference on Software Reuse.* IEEE Computer Society Press.

Halmans, G., and K. Pohl. 2003. Communicating the Variability of a Software-Product Family to Customers. *Journal of Software and Systems Modeling*: Springer.

Han, J. 2001. TRAM: A Tool for Requirements and Architecture Management. Pages 60-68. *the Australasian Computer Science Conference.* IEEE Computer Society, Gold Coast, Queensland, Australia.

Haumer, P., P. Heymans, M. Jarke, and K. Pohl. 1999. Bridging the Gap Between Past and Future in RE: A Scenario-Based Approach. *the Fourth IEEE International Symposium on Requirements Engineering (RE'99)*, University of Limerick, Ireland.

Haumer, P., M. Jarke, K. Pohl, and K. Weidenhaupt. 2000. Improving reviews of conceptual models by extended traceability to captured system usage. *Interacting with Computers Journal* 13: 77-95.

Haumer, P., K. Pohl, and K. Weidenhaupt. 1998. Requirements Elicitation and Validation with Real World Scenes. *IEEE Transactions on Software Engineering (TSE), Special Issue on Scenario Management* 24: 1036-1054.

Hayes, J. H., A. Dekhtyar, and J. Osborne. 2003. Improving requirements tracing via information retrieval. Pages 138-147. *11th IEEE International Conference on Requirements Engineering.* IEEE Computer Society, Washington, DC, USA.

Hayes, J. H., A. Dekhtyar, S. K. Sundaram, and S. Howard. 2004. Helping Analysts Trace Requirements: An Objective Look *12th IEEE International Conference on Requirements Engineering.*

Hull, E., K. Jackson, and J. Dick. 2002. *Requirements Engineering.* Springer-Verlag, London.

IEC. 1999. Functional Safety: Safety-Related Systems. International Standard IEC (International Electrical Commission) 61508.

IEEE-830. 1998. IEEE Recommended Practice for Software Requirements Specifications. IEEE Standard 830-1998.

Jacobson, I. 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Professional.

Jacobson, I., M. Griss, and P. Jonsson. 1997. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison-Wesley Professional.

Jarke, M. 1998. Requirement Tracing, Association for Computing Machinery. *Association for Computing Machinery. Communications of the ACM* 41.

JavaBeans. from http://java.sun.com/products/javabeans/

Jazayeri, M., A. Ran, and F. V. D. Linden. 2000. *Software Architecture for Product Families: Principles and Practice.* Addison-Wesley Pub (Sd).

Jirapanthong, W. 2004. Towards a Traceability Approach for Product Family Systems. *International Software Product Lines Young Researchers Workshop in International Software Product Line Conference,* Boston, MA.

Jirapanthong, W., and A. Zisman. 2004. Traceability for Product Family Systems: An XQuery Approach. *International Workshop on Requirements Reuse in System Family Engineering in International Conference on Software Reuse,* Madrid, Spain.

—. 2005. Supporting Product Line Development through Traceability. *12th Asia-Pacific Software Engineering Conference (APSEC 2005)*, Taipei, Taiwan.

—. 2006. XTraQue: Traceability for Product Line Systems. *Software and Systems Modeling* (under review).

John, I., and D. Muthig. 2002. Tailoring Use Cases for Product Line Modeling. *REPL'02*, Essen, Germany.

Jones, D. A., J. F. Nallon, D. M. York, and J. Simpson. 1995. Factors Influencing Requirement Management Toolset Selection. *the 5th Annual International Symposium of the INCOSE*. INCOSE, St. Louis, USA.

Kaindl, H. 1992. The Missing Link in Requirements Engineering. *Software Engineering Notes* June: 498-510.

Kang, K., S. Cohen, J. Hess, W. Novak, and A. Peterson. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Kang, K. C., S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. 1998. FORM: a feature-oriented reuse method with domain-specific architectures. *Annals of Software Engineering* 5: 143-168.

Keepence, B., and M. Mannion. 1999. Using Patterns to Model Variability in Product Families. *IEEE Software* 16: 102-108.

Kim, S. D., S. H. Chang, and H. J. La. 2005. Traceability Map: Foundations to Automate for Product Line Engineering. IEEE. Pages 274-281. *3rd ACIS International Conference on Software Engineering Research, Management & Applications(SERA05)*.

Knethen, A. v. 2002a. Automatic Change Support Based on a Trace Model. *the 1st International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE' 02)*, Edinburgh.

—. 2002b. Change-Oriented Requirements Traceability. Support for Evolution of Embedded Systems. *the International Conference on Software Maintenance (ICSM' 02)*.

Knethen, A. v., B. Paech, F. Kiedaisch, and F. Houdek. 2002. Systematic Recycling through Abstraction and Traceability. Pages 273-282. *IEEE Joint Int. Requirements Engineering Conference*. IEEE Computer Society, Essen, Germany.

Kotonya, G., and I. Sommerville. 1998. *Requirements Engineering, Process and Techniques*. John Wiley & Sons.

Krueger, C. W. 2001. Software Mass Customization. http://www.biglever.com/papers/BigLeverMassCustomization.pdf.

Lago, P., E. Niemela, and H. V. Vliet. 2004. Tool Support for Traceable Product Evolution. Pages 261-269. *the Eight European Conference on Software Maintenance and Reengineering (CSMR)*, Tampere, Finland.

Lawrence-Pfleeger, S., and S. Bohner. 1990. A Framework for Software Maintenance Metrics. *IEEE Conference on Software Maintenance.*

Lee, K., K. C. Kang, W. Chae, and B.W. Choi. 2000. Feature-based Approach to Object-Oriented Engineering of Applications for Reuse. *Software-Practice and Experience* 30: 1025-1046.

Leech, G., R. Garside, and M. Bryant. 1994. CLAWS4: The Tagging of the British National Corpus. Pages 622-628. *the 15th International Conference on Computational Linguistics (COLING 94)*, Kyoto, Japan.

Leishman, T. R., and D. A. Cook. 2002. Requirements Risks Can Drown Software Projects. *STSC CrossTalk* April.

Leite, J. C. S. d. P., and K. K. Breitman. 2003. Experience Using Scenarios to Enhance Traceability. *the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'03).* Montreal, Canada.

Letelier, P. 2002. A Framework for Requirements Traceability in UML-based Projects. *proceedings of the 1st International Workshop on Traceability for Emerging Forms of Software Engineering (TEFSE' 02)*, Edinburgh, UK.

Linden, F. v. d., J. Bosch, E. Kamsties, K. K¨ans¨al¨a, and H. Obbink. 2004. Software Product Family Evaluation. Pages 110-129. *the Third International Software Product Line Conference, SPLC 2004.* Springer Boston, MA, USA.

Lindvall, M., and K. Sandahl. 1996. Practical Implications of Traceability. *Software Practice and Experience* 26: 1161-1180.

Lindvall, M., and K. Sandahl. 1998. Traceability Aspects of Impact Analysis in Object-Oriented Systems. *Software Maintenance Research and Practice* 10: 37-57.

Lock, S., A. Rashid, P. Sawyer, and G. Kotonya. 1999. Systematic Change Impact Determination in Complex Object Database Schemata. Pages 31-40. *ECOOP.*

Maletic, J. I., and A. Marcus. 2001. Supporting Program Comprehension Using Semantic and Structural Information. *ICSE.*

Maletic, J. I., E. V. Munson, A. Marcus, and T. N. Nguyen. 2003. Using a Hypertext Model for Traceability Link Conformance Analysis. Pages 47-54. *the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'03)*. Montreal, Canada.

Mannion, M., O. Lewis, H. Kaindl, G. Montroni, and J. Wheadon. 2000. Representing Requirements on Generic Software in an Application Family Model. Pages 153-169. *ICSR*.

Marcus, A., and J. I. Meletic. 2003. Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing. Pages 125-137. *the 25th IEEE/ACM International Conference on Software Engineering (ICSE'03)*, Portland, OR, USA.

MBSE. 1993. Model-Based Software Engineering. http://www.sei.cmu.edu/technology/mbse/is.html.

McMullen, L. W. 1996-1997. Requirements Management Technology Overview. *Report of INCOSE Tools Database Working Group 1996-1997*.

MDA. from http://www.omg.org/mda/.

Metacase. from http://www.metacase.com/papers/.

Meyer, B. 1998. *Object Oriented Software Construction*. Prentice-Hall.

Mohan, K., and B. Ramesh. 2002. Managing variability with Traceability in product and Service Families. *In proceedings of the 35th Hawaii International Conference on System Sciences*. IEEE.

Murphy, G. C., D. Notkin, and K. Sullivan. 1995. Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. *Third ACM SIGSOFT Symp. Foundations of Software Eng.* Oct: 18-28.

Murray, L., A. Griffifths, P. Lindsay, and P. Strooper. 2002. Requirements Traceability for Embedded Software - an Industry Experience Report. Pages 63-69. *In proceedings of the 6th IASTED Software Engineering and Applications conference (SEA 2002)*. ACTA Press

NASA. Preferred Reliability Practices: Independent Verification and Validation of Embedded Software. Pages Practice No. PD-ED-1228. Marshal Space Flight Centre.

Nokia. from http://www.forum.nokia.com/main.html.

Northrop, L. M. 2002. SEI's Software Product Line Tenets. *IEEE Software* 19: 32-40.

Nuseibeh, B., and S. Easterbrook. 2000. Requirements Engineering: A Roadmap, In: The Future of Software Engineering. *ACM-IEEE*: 37-46.

OMA. from www.openmobilealliance.org/.

Ommering, R. v., F. v. d. Linden, and J. Kramer. 2000. The Koala component model for consumer electronics software. *IEEE Computer* 33: 78-85.

OMT. from http://www.omg.org/.

Parnas, D. 1976. The Design and Development of Program Families. *IEEE Transactions on software engineering* SE-2.

Philips. from http://www.philips.com.

Pinheiro, F. 2000. Formal and Informal Aspects of Requirements Tracing. *Position paper in proceedings of 3rd Workshop on Requirements Engineering (III WER)*, Rio de Janeiro, Brazil.

Pinheiro, F. A. C., and J. A. Goguen. 1996. An Object-Oriented Tool for Tracing Requirements. *IEEE Software* 13: 52-64.

Plankl, J., and G. Bockle. 2001. Modeling Concepts for Product Families. *Requirements Modeling and Traceability*. ESAPS report.

Pohl, K. 1994. The Three Dimensions of Requirements Engineering: A Framework and Its Applications. *Information systems* 19: 243-258.

—. 1996a. PRO-ART: Enabling Requirements Pre-Traceability. Pages 76. *the 2nd International Conference on Requirements Engineering (ICRE '96)*, Colorado, USA.

—. 1996b. *Process-Centered Requirements Engineering*. John Wiley & Sons.

Pohl, K., and P. Haumer. 1995. HYDRA: A Hypertext Model for Structurig Informal Requirements Representations. *the 2nd workshop on Requirements Engineering: Foundation of Software Quality (REFSQ' 95)*. Augustinus, Aachen, Germany, Jyvaskyla, Finland.

Poritz, A. B. 1998. Hidden Markov Models: A Guide Tour. Pages 7-13. *International Conference on Acoustics, Speech and Signal Processing*. IEEE., New York.

PuLSE. from http://www.iese.fhg.de/PuLSE/.

QADA. from http://www.vtt.fi/ele/research/soh/projects/families/qada.htm.

Ramesh, B., and V. Dhar. 1992. Supporting Systems Development Using Knowledge Captured During Requirements Engineering. *IEEE Transactions in Software Engineering* June 1992: 498-510.

Ramesh, B., and M. Edwards. 1993. Issues in the Development of a Requirements Traceability Model. Pages 256-259. *International Symposium on Requirements Engineering.*

Ramesh, B., and M. Jarke. 2001. Towards Reference Models for Requirements Traceability. *IEEE Transactions on Software Engineering* 27: 58-93.

Ramesh, B., T. Powers, C. Stubbs, and M. Edwards. 1995a. Implementing Requirements Traceability: A Case Study. Pages 89-95. *the Second IEEE International Symposium on Requirements Engineering*, York, United Kingdom.

Ramesh, B., C. Stubbs, T. Powers, and M. Edwards. 1995b. Lessons Learned from Implementating Requirements Traceability. *STSC CrossTalk* April.

RationalRose. from http://www.vtt.fi/ele/research/soh/projects/families/qada.htm.

RDT. from http://www.vtt.fi/ele/research/soh/projects/families/qada.htm.

Redondo, R. P. D., M. L. Nores, J. J. P. Aris, A. F. Vilas, J. G. Duque, A. G. Solla, B. B. Martinez, and M. R. Cabrer. 2004. Supporting Software Variability by Reusing Generic Incomplete Models at the Requirements Specification Stage. Pages 1-10. *8th International Conference, ICSR 2004*, Madrid, Spain.

RequisitePro. from http://www.rational.com.

Richardson, J., and J. Green. 2003. Traceability through Automatic Program Generation. *2nd International Workshop on Traceability in Emerging Forms of Software Engineering*, Montreal, Canada.

—. 2004. Automating Traceability for Generated Software Artifacts. Pages 20-24 Sept. 2004 *19th International Conference on Automated Software Engineering*, Linz, Austria.

Riebisch, M., K. Böllert, D. Streitferdt, and I. Philippow. 2002. Extending Feature Diagrams with UML Multiplicities. *the 6th world conference on Integrated Design and Process Technology (IDPT 2002)*, Pasadena, CA.

Riebisch, M., and I. Philippow. 2001. Evolution of Product Lines Using Traceability. *Workshop on Engineering Complex Object-Oriented Systems for Evolution in OOPSLA 2001*, Tampa Bay, Florida, USA.

RTM. from www.chipware.com.

Sawyer, P., A. Colebourne, and Sommerville. 1993. The MOG user interface builder: a mechanism for integrating application and user interface. *Interacting with Computers* 5: 315-332.

Schmid, K., and M. Schank. 2000. PuLSE-BEAT -- A Decision Support Tool for Scoping Product Lines. Pages 65-75. *the International Workshop on Software Architectures for Product Families*. Springer-Verlag

Sherba, S. A., K. M. Anderson, and M. Faisal. 2003a. A Framework for Mapping Traceability Relationships. *the 2nd International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE 2003)*, Canada.

—. 2003b. A Framework for Mapping Traceability Relationships. *the Second International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE'03), In conjunction with the 18th IEEE International Conference on Automated Software Engineering*, Montreal, Quebec, Canada.

Simos, M. 1995. Organization Domain Modelling (ODM).

Sinnema, M., et al. 2004. COVAMOF: A Framework for Modeling Variability in Software Product Families. *the third international conferences, SPLC*.

SLATE. from http:// tdtech.com.

Socorro, A. 1993. Design, Implementation and Evaluation of a Declarative Object-Oriented Programming Language. *Computing Laboratory*. Oxford University, Oxford.

Sommerville, I. 2000. *Software Engineering*. Addison Wesley.

Sommerville, I., and P. Sawyer. 1997. *Requirements Engineering – A Good Practice Guide*. John Wiley & Sons, New York.

Spanoudakis, G., and A. Finkelstein. 1997. Overlaps among Requirement Specifications. *Workshop on Living with Inconsistency in ICSE 97* Boston, USA.

Spanoudakis, G., A. Finkelstein, and D. Till. 1999. Overlaps in Requirements Engineering. *Automated Software Engineering* 6: 171-198.

Spanoudakis, G., and A. Zisman. 2005. Software Traceability: A Roadmap. *Advances in Software Engineering and Knowdledge Engineering* 3, Recent Advances,: 273-7.

Spanoudakis, G., A. Zisman, E. Pérez-Miñana, and P. Krause. 2004. Rule-based Generation of Requirements Traceability Relations. *Journal of Systems and Software* 72: 105-127.

Staudenmayer, N. S., and D. E. Perry. 1996. Session 5: Key Techniques and Process Aspects for Product Line Development. *the 10th International Software Process Workshop.*

Streitferdt, D. 2001. Traceability for System Families. Pages 803-804. *ICSE 2001.*

Sutcliffe, A. G., and N. A. M. Maiden. 1998. The Domain Theory for Requirements Engineering. *IEEE Trans* 24: 174-196.

Svahnberg, M., and J. Bosch. 2000. Issues Concerning Variability in Software Product Lines. *the Third International Workshop on Software Architectures for Product Families.* Springer Verlag, Berlin Germany.

Svahnberg, M., J. Gurp, and J. Bosch. 2001. On the Notion of Variability in Software Product Lines. Pages 45-55. *the Working IEEE/IFIP Conference on Software Architecture (WICSA 2001).*

Szyperski, C. 1997. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Professional

TestDirector. from http://www.mercuryinteractive.com/products/testdirector.

Thiel, S., and A. Hein. 2002. Systematic Integration of Variability into Product Line Architecture Design. Pages 130 - 153 *the Second International Conference on Software Product Lines (SPLC2).* Springer-Verlag.

Toranzo, M., and J. Castro. 1999. A comprehensive traceability model to support the design of interactive systems. Pages 283-284. *ECOOP Workshops.*

Tracz, W., L. Coglianese, and P. Young. 1993. A domain-specific software architecture engineering process outline. *SIGSOFT Software Engineering Notes* 18: 40-49.

UML. from http://www.uml.org.

UK_Ministry_of_Defence. 1997. Def Stan 00-55: Requirements for Safety Related Software in Defence Equipment.

Visio. 2003. http://www.microsoft.com/office/visio/prodinfo/overview.mspx.

Volere. from www.volere.co.uk.

Watkins, R., and M. Neal. 1994. Why and How of Requirements Tracing. *IEEE Software* 11: 104 -106.

Webber, D., and H. Gomaa. 2002. Modeling variability with the variation point model. Pages 109-122. *International Conference on Software Reuse.* Springer Verlag.

Weidenhaupt, K., K. Pohl, M. Jarke, and P. Haumer. 1998. Scenario Usage in System Development. *IEEE Software* 15: 34 - 45

Weiss, D. 1995. Software Synthesis: The FAST Process. *the International Conference on Computing in High Energy Physics (CHEP)*, Rio de Janeiro, Brazil.

—. 1998. Commonality Analysis: A Systematic Process for Defining Families. *Second International Workshop on Development and Evolution of Software Architectures for Product Families.*

Weiss, D., and C. T. R. Lai. 1999. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison Wesley, Reading, MA.

Westhuizen, C. v. d., and A. v. d. Hoek. 2002. Understanding and Propagating Architecutural Changes. Pages 95-109. *WICSA.*

Wong, S. K. M., and Y. Y. Yao. 1991. A probabilistic inference model for information retrieval. *Information systems* 16: 301-321.

—. 1995. On Modeling Information Retrieval with Probabilistic Inference. *ACM Transactions on Information Systems* 13: 38-68.

WordNet. from http://wordnet.princeton.edu/.

xADL2.0. from http://www.isr.uci.edu/projects/xarchuci/.

xArch. from http://www.isr.uci.edu/projects/xarch/.

XMI. from http://www.omg.org/technology/document/xmi.html.

XMLToolKit. from http://www.alphaworks.ibm.com/tech/xmitoolkit.

XPath. from http://www.w3.org/TR/xpath.

XQuery. from http://www.w3.org/TR/xquery/.

XTraQue. XTraQue. http://www.soi.city.ac.uk/~aj406/XTraQue/

Zisman, A., G. Spanoudakis, E. Perez-Minana, and P. Krause. 2002a. Towards a Traceability Approach for Product Families Requirements. *the 3rd ICSE Workshop on Software Product Lines: Economics, Architectures, and Implications*, Orlando, USA.

—. 2002b. Tracing Software Requirements Artefacts. *The 2003 International Conference on Software Engineering Research and Practice (SERP' 03)*, Las Vegas, Nevada, USA.