MODEL DRIVEN SOFTWARE PRODUCT LINE ENGINEERING: SYSTEM VARIABILITY VIEW AND PROCESS IMPLICATIONS

Abel
Gómez
Llana

PhD thesis ■ March 2012 ■ Supervised by Dr. Isidro Ramos Salavert ■ Programa de doctorado en programación declarativa e Ingeniería de la programación

MODEL DRIVEN SOFTWARE PRODUCT LINE ENGINEERING: SYSTEM VARIABILITY VIEW AND PROCESS IMPLICATIONS

ABEL GÓMEZ LLANA

A thesis submitted for the degree of Doctor of Philosophy in Computer Science

Departamento de Sistemas Informáticos y Computación

Supervised by Prof. Isidro Ramos Salavert

UNIVERSIDAD
POLITECNICA
DE VALENCIA

March, 2012

SUPERVISOR

Dr. Isidro RAMOS SALAVERT — *Universidad Politécnica de Valencia, Spain*

TUTOR

Dr. José Ángel CARSÍ CUBEL — *Universidad Politécnica de Valencia, Spain*

EXTERNAL REVIEWERS

Dr. José Miguel TORO BONILLA — *Universidad de Sevilla, Spain*

Dr. João ARAÚJO — *Universidade Nova de Lisboa, Portugal*

Dr. Jordi CABOT SAGRERA — *École des Mines de Nantes, France*

THESIS DEFENSE COMITTEE MEMBERS

Dr. José Miguel TORO BONILLA — *Universidad de Sevilla, Spain*

Dr. José Hilario CANÓS CERDÁ — *Universidad Politécnica de Valencia, Spain*

Dr. Nicole LEVY — *Conservatoire National des Arts et Métiers, France*

Dr. João ARAÚJO — *Universidade Nova de Lisboa, Portugal*

Dr. Francisco RUIZ GONZÁLEZ — *Universidad de Castilla–La Mancha, Spain*

ABEL GÓMEZ LLANA, *Model Driven Software Product Line Engineering: System Variability View and Process Implications.* Programa de doctorado en Programación Declarativa e Ingeniería de la Programación. This work has been supported by the Spanish Government under the National Program for Research, Development and Innovation DYNAMICA TIC2003-07804-C05-01, MOMENT TIN2006-15175-C05-01 (META TIN2006-15175-C05-01), MULTIPLE TIN2009-13838, and the FPU fellowship program, ref. AP2006-00690. PhD thesis © 2012 Abel Gómez Llana. All rights are reserved. Cover photograph: *Chevrolet Corvette ZR1, 2009 LS96, 2L V-8 SC (LS9), Component Exploded View* © 2008 General Motors. Printed in Spain.

Venceréis,
porque tenéis sobrada fuerza bruta.

Pero no convenceréis,
porque para convencer hay que persuadir.

Y para persuadir necesitaréis algo que os falta:
razón y derecho en la lucha.

— Miguel de Unamuno, 1864 – 1936,
docente, filósofo y escritor de la generación del 98.
Salamanca, 12 de Octubre de 1936.

Por la razón, la cultura y la ciencia

Dedicado a mi hermana, Arantxa,
y a mis padres. Os quiero.

A mis yayos y mi gran familia.

A quienes ya no están,
a quienes aún están por venir.

ABSTRACT

Software Product Line Engineering (SPLE) is a software development technique that aims to apply the principles of industrial manufacturing to obtaining software applications: i. e., a Software Product Line (SPL) is used to build a family of products with common features and whose members, however, may have some distinguished features. To identify these commonalities and variabilities *a priori* maximizes the reuse, and reduces the costs and development time. In this context, to describe these relationships among products with enough expressiveness becomes the key to success.

In recent years Model Driven Engineering (MDE) has emerged as a paradigm that allows dealing with software artifacts with a high level of abstraction. As a result, SPLs can benefit greatly from the standards and tools that have emerged within the community of MDE.

However, a good integration between SPLE and MDE has not been achieved yet. As a consequence, the mechanisms for variability management are not expressive enough. Thus, it is not possible to deal with variability issues in an effective way in complex software development processes, where different views of a system, model transformations and code generation play an important role.

This thesis presents MULTI**PLE**, a framework and a tool which aims to integrate accurate and efficient variability management mechanisms (which are inherent to SPLs development) together with MDE techniques. MULTIPLE provides domain specific languages to specify different views of software systems. Among these views special emphasis has been placed on the variability view because it is crucial for the specification of a SPL. Precise mechanism of specification, instantiation, validation and verification are provided for this view. MULTIPLE also allows to implement complex software

development processes of using model transformations and code generation.

The MULTIPLE tool has been used in five case studies in areas as diverse as the development of families of expert systems, the analysis of a large SPL in an industrial environment, bioinformatics, software metrics and software architectures.

RESUMEN

La Ingeniería de Líneas de Productos Software —Software Product Line Engineerings (SPLEs) en inglés— es una técnica de desarrollo de software que busca aplicar los principios de la fabricación industrial para la obtención de aplicaciones informáticas: esto es, una Línea de productos Software —Software Product Line (SPL)— se emplea para producir una familia de productos con características comunes, cuyos miembros, sin embargo, pueden tener características diferenciales. Identificar *a priori* estas características comunes y diferenciales permite maximizar la reutilización, reduciendo el tiempo y el coste del desarrollo. Describir estas relaciones con la suficiente expresividad se vuelve un aspecto fundamental para conseguir el éxito.

La Ingeniería Dirigida por Modelos —Model Driven Engineering (MDE) en inglés— se ha revelado en los últimos años como un paradigma que permite tratar con artefactos software con un alto nivel de abstracción de forma efectiva. Gracias a ello, las SPLs puede aprovecharse en gran medida de los estándares y herramientas que han surgido dentro de la comunidad de MDE.

No obstante, aún no se ha conseguido una buena integración entre SPLE y MDE, y como consecuencia, los mecanismos para la gestión de la variabilidad no son suficientemente expresivos. De esta manera, no es posible integrar la variabilidad de forma eficiente en procesos complejos de desarrollo de software donde las diferentes vistas de un sistema, las transformaciones de modelos y la generación de código juegan un papel fundamental.

Esta tesis presenta MULTI**PLE**, un marco de trabajo y una herramienta que persiguen integrar de forma precisa y eficiente los mecanismos de gestión de variabilidad propios de las SPLs dentro de los procesos de MDE. MULTIPLE proporciona lenguajes específicos de dominio para especificar diferentes vistas de los sistemas software.

Entre ellas se hace especial hincapié en la vista de variabilidad ya que es determinante para la especificación de SPLs. Para esta vista se proporcionan mecanimos precisos de especificación, instanciación, validación y verificación. MULTIPLE permite, además, implementar procesos complejos de desarrollo de software empleando transformaciones de modelos y generación de código.

La herramienta MULTIPLE ha sido utilizado en cinco casos de estudio en ámbitos tan diferentes como el desarrollo de familias de sistemas expertos, el análisis de una SPL de gran tamaño en un ambiente industrial, la bioinformática, las métricas software o las arquitecturas software.

RESUM

L'Enginyeria de Línies de Productes de Programari —Software Product Line Engineerings (SPLEs) en anglès— és una tècnica de desenvolupament de programari que busca aplicar els principis de la fabricació industrial per a l'obtenció d'aplicacions informàtiques: és a dir, una Línia de Productes de Programari —Software Product Line (SPL)— s'empra per produir una família de productes amb característiques comuns, les quales, però, poden tenir característiques diferencials. Identificar *a priori* aquestes característiques comuns i diferencials permet maximitzar la reutilització, reduint el temps i el cost del desenvolupament. Descriure aquestes relacions amb la suficient expressivitat es torna un aspecte fonamental per aconseguir l'èxit.

L'Enginyeria Dirigida per Models —Model Driven Engineering (MDE) en anglès— s'ha revelat en els últims anys com un paradigma que permet tractar amb artefactes de programari amb un alt nivell d'abstracció de forma efectiva. Gràcies a això, les SPLs poden aprofitar-se en gran mesura dels estàndards i les eines que han sorgit dins de la comunitat de MDE.

No obstant això, encara no s'ha aconseguit una bona integració entre SPLE i MDE, i com a conseqüència, els mecanismes per a la gestió de la variabilitat no són prou expressius. Amb la qual cosa no és possible integrar la variabilitat de manera eficient en processos complexos de desenvolupament de programari on les diferents vistes d'un sistema, les transformacions de models i la generació de codi juguen un paper fonamental.

Aquesta tesi presenta MULTI**PLE**, un marc de treball i una eina que persegueixen integrar de forma precisa i eficient els mecanismes de gestió de variabilitat propis de les SPLs dins dels processos de MDE. MULTIPLE proporciona llenguatges específics de domini per especificar diferents vistes dels sistemes. Entre elles es fa especial èmfasi en

la vista de variabilitat, ja que és determinant per a la especificació de SPLs. Per a aquesta vista es proporcionen els mecanismes necessaris d'especificació, instanciació, validació i verificació. MULTIPLE permet, a més, implementar processos complexos de desenvolupament de programari emprant transformacions de models i generació de codi.

L'eina MULTIPLE ha estat utilitzada en cinc casos d'estudi en àmbits tan diversos com el desenvolupament de famílies de sistemes experts, l'anàlisi d'una SPL de grans dimensions en un entorn industrial, la bioinformàtica, les mètriques de programari o les arquitectures de programari.

«When you make the finding yourself
—even if you're the last person on Earth to see the light—
you never forget it»

— Carl Sagan
American Astronomer, Writer and Scientist, 1934-1996

AGRADECIMIENTOS

Me gustaría dar las gracias a todas aquellas personas que han colaborado, ya sea directa o indirectamente, en la realización de este trabajo, en el que he invertido tanto esfuerzo.

En primer lugar me gustaría expresar mi más profundo y sincero agradecimiento a Isidro, quien ha dirigido y tutelado esta tesis por encima de cualquier otra circunstancia. Muchas gracias por confiar en mí y en este trabajo; muchas gracias por la dedicación, experiencia, sabiduría, libertad y cercanía que me has ofrecido a lo largo de estos años.

Muchas gracias tambien a Pepe por tutorizar este trabajo cuando lo he necesitado, así como al resto de miembros del grupo ISSI: Silvia, Emilio, Javier, Patricio y Mª Carmen; y muy especialmente, a su director fundador, Isidro, como a su actual director, José Hilario. Gracias por ofrecerme una oportunad como investigador y gracias por cuidar, promover y mantener un excelente ambiente y grupo de trabajo. Gracias a Artur y Pepe, por guiar mi tarea como investigador en sus primeros años y proyectos. Y finalmente gracias a mis actuales compañeros de laboratorio: David, Adrián, Javier y Sonia por los ratos que pasamos a diario.

Mención especial merecen Alejandro, Cristóbal, Manolo y José Antonio, quienes más que compañeros de laboratorio son amigos. Con ellos he compartido mi día a día; hemos compartido comidas, cenas y cervezas; y hemos celebrado bodas y nacimientos. Me gustaría incluir en este especial agradecimiento a Javier, Elena y Jennifer; así como a compañeros con quienes compartí muchos momentos y que, aunque ya dejaron España, permanecen en mi recuerdo: Carlos, Nour, Nelly, Gonzalo, Isabel y Felipe.

Me gustaría dar las gracias a aquellos que han contribuido de forma fundamental al desarrollo de esta tesis. Especialmente me gustaría mostrar mi más sincera gratitud a Mª Eugenia. Muchísimas gracias por ofrecerme una vía de investigación y un caso de estudio fundamental para esta tesis. Gracias por tu generosidad para compartir tanto recursos como esfuerzos cuando el futuro de otras líneas de investigación era incierto, y gracias por continuar el trabajo conmigo desde el lejano México. Muchas gracias también a Rogelio, quien en su tesis proporciona una base fundamental para este trabajo. No pueden faltar en este agradecimiento todos aquellos que han participado directamente en esta tesis, ya sea en proyectos conjuntos; proporcionando los casos de estudio; realizando sus respectivas tesis, tesis de máster o proyectos finales de carrera; o ayudando en tareas de implementación: Prof. Hans-Dieter Ehrich, Félix O. García, María Gómez, Clara Khachan, Beatriz Mora, Elena Navarro, Francisco Ruiz, Claudia Täubner y Mareike Ziegler. Quisiera además recordar también tanto a excompañeros como investigadores con quienes

he compartido discusiones, proyectos, conferencias y momentos de asueto, tales como Pascual Queral, Joaquín Oriente, Luis Hoyos o Carlos Cuesta.

Finalmente, me gustaría dar las gracias a todos aquellos que habéis contribuido a este trabajo desde fuera del ambiente académico. Gracias a mis amigos y mi familia por hacer que todo el esfuerzo tenga un sentido. Gracias a Lorena, Rafa, (y Helena), Alberto, Ana, Bigorra…; gracias a Arantxa y a mis padres, Abel y Mª Carmen; gracias a mis yayos, Abel y Fermina; gracias a mis primos, Ana, Carlos, Cristina, Encarna, Lorena y María(s); y gracias a mis tíos, Carlos, Encarna, Feli, Isabel, Jesús, Nieves, Rafa, Rosa y Rosi.

Gracias a todos.

CONTENTS

I INTRODUCTION 1
1 INTRODUCTION 3
1.1 Motivation 4
1.2 Objectives 5
1.3 Structure 6
II PRELIMINARIES AND STATE OF THE ART 9
2 MDE: AUTOMATING CODING IN SOFTWARE DEVELOPMENT 13
2.1 Model Driven Engineering Open standards 15
2.2 Model Driven Architecture 16
2.3 Meta Object Facility 18
2.4 Object Constraint Language 20
2.4.1 Language features 21
2.5 Query/View/Transformation 22
2.5.1 Languages 23
2.5.2 The *Relations* language 25
2.6 Summary 29
3 SUPPORTING TECHNOLOGIES FOR MDE 31
3.1 The Eclipse Platform 32
3.1.1 Eclipse Modeling Framework 33
3.1.2 Graphical Modeling Framework 38
3.1.3 Model Development Tools 40
3.2 MOMENT: A framework for Model Management 41
3.3 ATLAS Transformation Language 41
3.4 IBM Model Transformation Framework 42
3.5 MediniQVT 42
3.6 Summary 44
4 SOFTWARE PRODUCT LINES 45
4.1 Software Product Line Engineering 46

4.2 Describing variabilities and commonalities 48
4.2.1 Introduction 51
4.2.2 Classic feature models 51
4.2.3 FeatuRSEB and PLUSS feature models . . 54
4.2.4 Cardinality-based feature models 56
4.2.5 Feature model configurations 58
4.3 Summary 59

III VARIABILITY VIEW ON MULTI-MODEL DRIVEN SOFTWARE PRODUCT LINES 61

5 MULTI-MODEL DRIVEN PRODUCT LINE ENGINEERING 65
5.1 System views and the multi-model 67
5.2 Views, models and metamodels in MULTIPLE . . 69
5.3 Summary 70

6 FEATURE MODEL CONFIGURATION ISSUES 73
6.1 Introduction 74
6.2 Feature models, configurations and MOF 74
6.3 Describing feature model configurations as instances 79
6.4 Summary and conclusions 81

7 USING FEATURE MODELS IN MDE PROCESSES 83
7.1 Process overview 83
7.2 Cardinality-based feature metamodel 85
7.2.1 Feature models structure 85
7.2.2 Feature model constraints 86
7.2.3 Cardinality-based feature metamodel in MOF 87
7.3 The Domain Variability Model 90
7.3.1 The structure of the DVM 91
7.3.2 Constraints over the DVM 100
7.4 Feature model configurations 109
7.5 Summary and conclusions 112

IV THE MULTIPLE FRAMEWORK AND MMDSPL DEVELOPMENT AND ANALYSIS 113

8 THE MULTIPLE FRAMEWORK 117

8.1 Subsystems and components overview 118
8.2 The Eclipse Platform 123
8.3 Built-in metamodels 125
8.3.1 Variability metamodel support 125
8.3.2 FAMA metamodel support 138
8.3.3 Modular metamodel support 146
8.3.4 Component–connector metamodel support 154
8.3.5 PRISMA metamodel support 161
8.4 Transformations Subsystem 166
8.4.1 QVT engine 167
8.4.2 QVT transformation invocation model support 168
8.4.3 Traceability metamodel support 171
8.4.4 QVT Launcher 178
8.4.5 QVT Command-line interface 187
8.5 Validation Subsystem 194
8.5.1 OCL Support 195
8.5.2 OCL Support CLI 203
8.5.3 Variability Model Checking 206
8.6 MULTIPLE EMF Utils 214
8.6.1 EMOF Converter utility 214
8.6.2 Register EMF utility 214
8.6.3 Registry viewer utility 216
8.7 Summary and conclusions 216
9 MMDSPL FOR DIAGNOSTIC EXPERT SYSTEMS DEVELOPMENT 219
9.1 Technological spaces 221
9.2 Field Study: Diagnostic Expert Systems 222
9.2.1 Diagnostic Expert Systems Reference Architecture 223
9.2.2 Diagnostic Expert Systems Structural Variability 224
9.2.3 Diagnostic Expert Systems Behavioral Variability 225

9.2.4 Diagnostic Expert Systems Application Domain Variability 226
9.2.5 Conclusions 227
9.3 BOM initial proposal: BOM–Eager 228
9.3.1 Variability management in BOM 228
9.3.2 Software system views in BOM–Eager . . 233
9.3.3 Relationships among system views 234
9.3.4 Modeling the BOM approach 236
9.3.5 BOM–Eager implementation 239
9.4 Turning BOM into a MMDSPLE process 242
9.4.1 Representing the first variability in BOM–Lazy 242
9.4.2 Software system views in BOM–Lazy . . . 244
9.4.3 Relationships among metamodels 245
9.5 BOM–Lazy implementation 248
9.5.1 T1 transformation 249
9.5.2 T2 transformation 260
9.6 Summary and conclusions 268
10 AUTOMATED ANALYSIS OF FEATURE MODELS IN MULTIPLE 273
10.1 Context and Motivation 274
10.2 Case study 276
10.2.1 Feature modeling in Rolls-Royce plc . . . 277
10.2.2 Analysis process overview 278
10.2.3 Source Model structure 281
10.2.4 A step by step description of the process . 282
10.3 Interpreting the obtained results 293
10.3.1 Syntactic analysis 293
10.3.2 Semantic Analysis 297
10.3.3 Conclusions about the analysis results . . 300
10.3.4 Efficiency and limitations of the automated analysis tool 302
10.4 Summary and conclusions 303

V THE MULTIPLE FRAMEWORK IN 3RD PARTY PROJECTS AND TOOLS 305
11 BIOLOGICAL DATA MIGRATION USING MULTIPLE 309
11.1 Case study 311
11.1.1 Toll-like receptors and the TLR4 signal transduction pathway 313
11.1.2 An approach to the study of the TLR4 signal transduction pathway 315
11.2 A MDSD approach in biological data migration . . 317
11.2.1 Architecture and overview of the tool . . 318
11.2.2 Development of the source and the target models 319
11.2.3 Transformation process 323
11.3 Running example 324
11.3.1 Result files 327
11.3.2 Result file in *CPN Tools* 328
11.4 Conclusions 329
12 SOFTWARE MEASUREMENT BY USING QVT TRANSFORMATIONS 333
12.1 Related works 335
12.2 Software Measurement Framework 336
12.2.1 Conceptual architecture 337
12.2.2 Technological aspects 339
12.2.3 Method 342
12.3 Example 343
12.4 Conclusions 347
13 MORPHEUS: A TOOL FOR THE ATRIUM METHODOLOGY 349
13.1 ATRIUM at a glance 350
13.2 MORPHEUS: a MDD supporting tool 353
13.2.1 Requirements Environment 354
13.2.2 Scenario Environment 357
13.2.3 Software Architecture Environment . . . 361
13.3 Related works 363
13.4 Conclusions and further works 364

VI CLOSURE 365
14 RELATED WORKS 369
14.1 MULTIPLE feature models and other feature modeling proposals 370
14.2 MULTIPLE and the OMG CVL 371
14.3 Feature models and class diagrams 371
14.4 Feature model constraints 372
14.5 Feature models and other SPL approaches 373
15 SUMMARY AND CONCLUSIONS 377
16 PUBLICATIONS 385

APPENDICES 393
A TRANSFORMATION FEATURES2CLASSDIAGRAM 395
B TRANSFORMATION MODULES2COMPONENTS 407
C TRANSFORMATION COMPONENTS2PRISMA 413
D TRANSFORMATION MULTIPLEFEATURES2FAMAFEATURES 421
E FAMA XML SCHEMA DEFINITION 425
F RUNNING A QVT TRANSFORMATION USING MEDINI QVT 429
G TRANSPATH2CPN TRANSFORMATION 435

BIBLIOGRAPHY 445

LIST OF FIGURES

Figure 2.1 MOF layers 19
Figure 2.2 Relationships among the QVT metamodels . 24

Figure 3.1 Simplified version of the *Ecore* metamodel . 35
Figure 3.2 GMF workflow overview 39

Figure 4.1 Essential Activities for Software Product Lines 48
Figure 4.2 Feature model genealogy 50
Figure 4.3 Example of FODA relationships 52
Figure 4.4 Example of the *implies* and *excludes* relationships 53
Figure 4.5 *XOR* and *OR* groups in *FeatuRSEB* 54
Figure 4.6 Example PLUSS feature model 55
Figure 4.7 Text editor configuration example 57
Figure 4.8 Example of a feature model and the two possible configurations that it represents 58

Figure 5.1 Multi-Modeling Driven Software Product Line Engineering 67

Figure 6.1 Configuration of FODA feature models in MOF 75
Figure 6.2 Staged configuration through specialization . 76
Figure 6.3 Configuration through specialization in the context of MOF 76
Figure 6.4 Example of a model transformation 77
Figure 6.5 Definition and configuration of feature models in the context of MOF 78
Figure 6.6 EMF and the four-layer architecture of MOF . 80
Figure 6.7 Dealing with feature model configurations in EMF 81

Figure 7.1 Feature models and MDE: process overview . 84

Figure 7.2 Cardinality-based features metamodel 88
Figure 7.3 Example cardinality-based feature model . . 90
Figure 7.4 *Feature2Class* relation 92
Figure 7.5 *FeatureAttribute2ClassAttribute* relation . . . 93
Figure 7.6 *StructuralRealtionship2Reference* relation . . 94
Figure 7.7 *Group2Reference* relation 94
Figure 7.8 *GroupChild2Classes* relation 96
Figure 7.9 Cardinalities in feature groups 96
Figure 7.10 *GroupChild2ChildrenAnnot* relation 97
Figure 7.11 *GroupChild2LowerAnnot* relation 99
Figure 7.12 *GroupChild2UpperAnnot* relation 100
Figure 7.13 *ExcludesRelationship2Modelconstraint* relation 102
Figure 7.14 *ImpliesRelationship2ModelConstraint* relation 103
Figure 7.15 *BiconditionalRelationship2ModelConstraint* relation 104
Figure 7.16 *UsesRelationship2Reference* relation 105
Figure 7.17 *UsesRelationship2EOppositeEreference* relation 106
Figure 7.18 *FMCLConstraint2OCLConstraint* relation . . 108
Figure 7.19 Generated class diagram for the example cardinality-based feature model 109
Figure 7.20 A valid configuration of the example DVM . . 111

Figure 8.1 Architecture of the MULTIPLE framework . 121
Figure 8.2 Features metamodel represented in an *Ecore* tree editor 127
Figure 8.3 Workflow followed to obtain the features model editor 130
Figure 8.4 *Gmftool* model used to generate the GMF-based feature model editor 131
Figure 8.5 *Gmfgraph* model used to generate the Graphical Modeling Framework (GMF)-based feature model editor 132
Figure 8.6 *Gmfmap* model used to generate the GMF-based feature model editor 133
Figure 8.7 New feature model wizard 134

Figure 8.8 Standard tree editor for feature models . . . 135
Figure 8.9 Initialize features diagram file 136
Figure 8.10 New feature diagram wizard 136
Figure 8.11 Example model and diagram files to represent a feature model graphically 137
Figure 8.12 Example feature model 138
Figure 8.13 FAMA metamodel support initial files 140
Figure 8.14 *Model2Model* relation 142
Figure 8.15 *StructuralRelationship2BinaryRelation* relation 143
Figure 8.16 *Group2SetRelation* relation 143
Figure 8.17 *ExcludesRelationship2ExcludesType* relation . 145
Figure 8.18 ImpliesRelationship2RequiresType relation . 145
Figure 8.19 Modular view metamodel 147
Figure 8.20 *Gmfgraph* model used to generate the GMF-based modular model editor 150
Figure 8.21 *Gmftool* model used to generate the GMF-based modular model editor 151
Figure 8.22 *Gmfmap* model used to generate the GMF-based modular model editor 151
Figure 8.23 New modular model wizard 152
Figure 8.24 GMF-based modular view editor 153
Figure 8.25 Component-connector view metamodel . . . 155
Figure 8.26 *Gmfgraph* model used to generate the GMF-based component-connector model editor . 157
Figure 8.27 *Gmftool* model used to generate the GMF-based component-connector model editor . 158
Figure 8.28 *Gmfmap* model used to generate the GMF-based component-connector model editor . 158
Figure 8.29 Software architecture of the Agrobot 160
Figure 8.30 PRISMA metamodel 162
Figure 8.31 QVT transformation invocation model . . . 169
Figure 8.32 MULTIPLE traceability metamodel 172
Figure 8.33 Navigating from a source element in the traceability editor 175

Figure 8.34 Navigating from a *traceability link* in the traceability editor 176
Figure 8.35 Properties view of a domain element 176
Figure 8.36 Properties view of a traceability link 177
Figure 8.37 Properties view of a traceability link 177
Figure 8.38 *Run configurations* menu 183
Figure 8.39 *Edit configuration* dialog 184
Figure 8.40 *Run as QVT Transformation* contextual menu 184
Figure 8.41 *Edit configuration and launch* dialog window 185
Figure 8.42 Setting the input and output models of a model transformation 186
Figure 8.43 QVT transformation ready to be executed . . 186
Figure 8.44 Progress monitor of a model transformation 187
Figure 8.45 Result files of the example model transformation 187
Figure 8.46 CLI invocation model 189
Figure 8.47 Example of a model transformation using the CLI engine 192
Figure 8.48 Example of a model transformation using a compiled metamodel and the CLI engine . . 194
Figure 8.49 Example *car.ecore* model with OCL constraints 196
Figure 8.50 Generating an OCL textual file 198
Figure 8.51 Generated OCL textual file 199
Figure 8.52 Creating an instance of an *Ecore* model dynamically 200
Figure 8.53 Example of a incorrectly defined instance . . 201
Figure 8.54 *MULTIPLE OCL checker* contextual menu . . 201
Figure 8.55 *MULTIPLE OCL checker*: unsuccessful check 202
Figure 8.56 *MULTIPLE OCL checker*: unsuccessful check details 202
Figure 8.57 Example of a correctly defined instance . . . 203
Figure 8.58 Example execution of the Object Constraint Language (OCL) Command-Line Interface (CLI) engine 206
Figure 8.59 Sample void feature model 209

Figure 8.60 Sample void feature model represented in FAMA 210
Figure 8.61 *Detect and explain errors* contextual menu . . 211
Figure 8.62 Result of *detect and explain errors* 211
Figure 8.63 *To FAMA feature model* (as text) contextual menu 212
Figure 8.64 Sample feature model in the FAMA textual representation 212
Figure 8.65 *To FAMA feature model* (as text) contextual menu 213
Figure 8.66 Sample feature model in the FAMA textual representation 213
Figure 8.67 *Metamodels* view and *Metamodel tree editor* . 215

Figure 9.1 Reference Architecture of Expert Systems . . 223
Figure 9.2 Medical diagnosis use case diagram and its corresponding base architecture 225
Figure 9.3 Graph describing the inference processes for medical diagnosis and educational diagnosis 226
Figure 9.4 Features of the first and second variability of our SPL 230
Figure 9.5 Original Feature Model of the first variability in BOM 230
Figure 9.6 The domain conceptual model 231
Figure 9.7 The application domain conceptual model . 232
Figure 9.8 Variability management and system views in BOM–Eager 234
Figure 9.9 Binary Decision Tree to select a skeleton architecture for a DES 235
Figure 9.10 The baseline 237
Figure 9.11 Production plan through BOM–Eager approach 238
Figure 9.12 Feature Model of the first variability in BOM 243
Figure 9.13 The domain conceptual model 243
Figure 9.14 The application domain conceptual model . 244
Figure 9.15 The T1 and T2 model transformations in BOM 246

Figure 9.16 Production plan through BOM–Lazy approach 248
Figure 9.17 The BOM–Lazy architecture 249
Figure 9.18 *ModulesModel2ComponentsModel* relation . 254
Figure 9.19 *UseCase2Connector* relation 255
Figure 9.20 *Module2Component* relation 256
Figure 9.21 *Module2RolePort* relation 258
Figure 9.22 *ConnectRoleAndPort* relation 258
Figure 9.23 *Function2Service* relation 259
Figure 9.24 *Function2Relation* relation 259
Figure 9.25 *CCModel2PRISMAArchitecture* relation . . . 260
Figure 9.26 *Component2Component* relation 261
Figure 9.27 *Port2PortInterface* relation 262
Figure 9.28 *Component2FunctionalAspect* relation 262
Figure 9.29 *Property2ConstantAttribute* relation 263
Figure 9.30 *Hypotheses2VariableAttribute* relation 264
Figure 9.31 *Rule2DerivedAttribute* relation 264
Figure 9.32 *Property2Parameter* relation 264
Figure 9.33 *Hypotheses2Parameter* relation 264
Figure 9.34 *Connector2ConnectorPortInterface* relation . 265
Figure 9.35 *ConnectRolePort* relation 266
Figure 9.36 *Connector2CoordinatorAspect* relation 266
Figure 9.37 *AddServices2Interface* relation 267
Figure 9.38 *AddPlayedRole2Aspect* relation 268
Figure 9.39 *AddPlayedRole2Port* relation 268
Figure 9.40 *AddAttachmentsBindingsToPortAndArc* relation 268

Figure 10.1 Basic components of a gas turbine engine . . 276
Figure 10.2 Cardinality-based feature model equivalent to the one shown in Fig. 4.6 277
Figure 10.3 Schema of the proposal 279
Figure 10.4 Rolls-Royce PLUSS feature model 283
Figure 10.5 Running the PLUSS parser 284

Figure 10.6 The Rolls-Royce feature model shown in the MULTIPLE feature modeling tree editor . . 285
Figure 10.7 Console showing the syntactical analysis results 286
Figure 10.8 Dialog box showing the the *MultipleFeatures-2FamaFeatures* transformation is ready to be executed 287
Figure 10.9 Rolls-Royce model represented as a FAMA model in the FAMA tree editor 288
Figure 10.10 Rolls-Royce model represented using the FAMA XML serialization format 289
Figure 10.11 Running FAMA analysis from the MULTIPLE user interface 290
Figure 10.12 Calculating the number of products using FAMA 291
Figure 10.13 Detecting errors using FAMA 291
Figure 10.14 Calculating products using FAMA 292
Figure 10.15 Empty features in the original feature model and a possible solution 293
Figure 10.16 Duplicated features in the original feature model and a possible solution 294
Figure 10.17 Correct representation using cardinality-based notation 296
Figure 10.18 Example of ambiguous use of feature groups and a possible solution 297
Figure 10.19 Invalid relationship and a possible solution . 298
Figure 10.20 Example of a false-mandatory feature 299
Figure 10.21 Example of dead features and two possible solutions 300

Figure 11.1 Signal Transduction 311
Figure 11.2 TLR4 signal transduction pathway in the TRANSPATH® database 313
Figure 11.3 Petri net example 316
Figure 11.4 Architecture of the tool 318
Figure 11.5 Model of the TRANSPATH® database 320

Figure 11.6 Model of the *CPN Tools* application 322
Figure 11.7 Partial representation of the TLR4 signal transduction pathway in *CPN Tools* 325
Figure 11.8 Workspace with the example files 326
Figure 11.9 Actual contents of the `example.xml` file . . . 327
Figure 11.10 Transpath2CPN transformation ready to be executed 327
Figure 11.11 Result files 328
Figure 11.12 Editor for *cpn* models 329
Figure 11.13 Export to *CPN Tools* 330
Figure 11.14 Content of the final XML file 331
Figure 11.15 Final result shown in *CPN Tools* 331

Figure 12.1 Conceptual framework with which to manage software measurement 338
Figure 12.2 Elements of the FMESP adaptation in a MDA context 340
Figure 12.3 QVT–Relations transformation model 341
Figure 12.4 Software Measurement Process 342
Figure 12.5 Relationship between Relational Database Metamodel and SMM 344
Figure 12.6 Relational Database model (relational schema) 345

Figure 13.1 An outline of ATRIUM 351
Figure 13.2 Main architecture of MORPHEUS 354
Figure 13.3 Main elements of the requirements environment 355
Figure 13.4 Meta-Modeling work context of the MORPHEUS Requirements Environment 356
Figure 13.5 Core-metamodel for the requirements environment 356
Figure 13.6 Describing a new meta-artifact in MORPHEUS 357
Figure 13.7 Modelling work context of the MORPHEUS Requirements Environment 358

Figure 13.8 Main elements of the *Scenarios Environment* 358
Figure 13.9 What the Scenario Editor looks like 359
Figure 13.10 Describing the Synthesis processor 360
Figure 13.11 Generate architecture dialog 360
Figure 13.12 Main elements of the Software Architecture Environment 361
Figure 13.13 What the Architectural Editor looks like . . . 362

Figure 15.1 Extended architecture of the MULTIPLE framework 381

LIST OF TABLES

Table 4.1 Development and application of a SPL 49
Table 4.2 Cardinality-based feature modeling basic primitives 56
Table 4.3 Summary of feature model notations 60

Table 7.1 Cardinality-based feature metamodel: proposed types of relationships between features 85
Table 7.2 Summary of transformation patterns from FMCL to OCL 107

Table 8.1 Icons of the features metamodel elements . . 129
Table 8.2 Correspondences between Cardinality-based feature models and FAMA 142

Table 9.1 Example of a skeleton and PRISMA type aspects 241
Table 9.2 Rules and involved elements in the T1 and T2 transformations 272

Table 10.1 Correspondences between Cardinality-based feature models and PLUSS 278
Table 10.2 Source model extract 280
Table 10.3 Example of features incorrectly defined in the source CSV file 296
Table 10.4 Percentages of correction grouped by element type 301

Table 11.1 Mappings between the source and the target domain 323

LISTINGS

Listing 4.1 Textual representation for the FODA *excludes* and *requires* relationships 53

Listing 7.1 *buildGroupConstraint* OCL query 98
Listing 7.2 *toString* OCL query 98
Listing 7.3 *translateFMCLtoOCL* OCL query 108
Listing 7.4 Generated OCL expression for the example feature model 110

Listing 8.1 *evaluateQvt* method 181
Listing 8.2 Usage of the CLI of the Query/View/Transformation (QVT) engine 190
Listing 8.3 Example invocation file: uml2rdbmsconfig.xmi 191
Listing 8.4 Example invocation file: uml2rdbmsconfig-mod.xmi 193
Listing 8.5 Usage of the CLI of the OCL engine 205
Listing 8.6 Sample void feature model represented in FAMA-Extensible Markup Language (XML) . 209

Listing 9.1 Sample instances of the Application Domain Conceptual Model (ADCM) 233
Listing 9.2 Example code generated by the PRISMA-MODEL-COMPILER 240
Listing 9.3 *getComponentName* OCL query 257

Listing 12.1 Relation domain elements from extended QVT–Relations model 346
Listing 12.2 Function elements from extended QVT–Relations model 346
Listing 12.3 Measurement result 347

Listing A.1 Full Features2ClassDiagram transformation . 395

Listing B.1 Full Modules2Components transformation . 407

Listing C.1 Full Components2Prisma transformation . . 413

Listing D.1 MULTIPLE Features to FAMA features transformation 421

Listing E.1 FAMA XML Schema Definition 425

Listing F.1 Running a QVT transformation programmatically using medini QVT:*QvtTransformationJob* class 429

Listing G.1 Full Transpath2CPN transformation 435

ACRONYMS

ADCM	Application Domain Conceptual Model xxxiii
ADL	Architecture Description Language 240
AO-MD-SPL	Aspect-Oriented Model-Driven Software Product Lines 374
AOSD	Aspect-Oriented Software Development 161
AOP	Aspect-Oriented Programming 375
API	Application Programming Interface 40
AST	Abstract Syntax Tree 170
ATL	ATLAS Transformation Language 41
ATRIUM	Architecture generaTed from RequIrements applying a Unified Methodology 7
BDT	Binary Decision Tree 235
BOM	Baseline Oriented Modeling 115
BPMN	Business Process Model and Notation 40
CASE	Computer Aided Software Engineering 13
CBSD	Component-Based Software Development 161
CD14	Cluster of Differentiation 14 314
CIM	Computational Independent Model 236
CLI	Command-Line Interface xxiv
CMMI	Capability Maturity Model Integration 335
CMOF	Complete Meta-Object Facility 40
CORBA	Common Object Request Broker Architecture . . 15
CORE	Computing Research and Education Association of Australasia 387

CSV	Comma-Separated Values 279
CVL	Common Variability Language 371
DCM	Domain Conceptual Model 229
DES	Diagnostic Expert Systems 222
DOM	Document Object Model 41
DOORS	Dynamic Object-Oriented Requirements System 279
DSL	Domain Specific Language 6
DTD	Document Type Definition 316
DVM	Domain Variability Model 80
ECSIT	Evolutionarily Conserved Signaling Intermediate in Toll pathways 314
EFTCoR	Environmental Friendly and Cost-Effective Technology for Coating Removal 352
EMF	Eclipse Modeling Framework 33
EMOF	Essential Meta-Object Facility 24
EPL	Eclipse Public License 42
ES	Expert Systems 219
FAMA	FeAture Model Analyser 122
FMCL	Feature Modeling Constraint Language 87
FMESP	Framework for the Modeling and Evaluation of Software Processes 334
FMP	Feature Modeling Plugin 370
FM	Feature Model 84
FODA	Feature-Oriented Domain Analysis 51
FOM	Feature Oriented Modeling 235
FOP	Feature Oriented Programming 242
FQN	Fully Qualified Name 189

GEF Graphical Editing Framework 38

GMF Graphical Modeling Framework. xxii

GUI Graphical User Interface 38

HCI Human-Computer Interaction 251

IDE Integrated Development Environment 32

IEEE Institute of Electrical and Electronics Engineers 51

IKK IκB Kinase 314

IMM Information Management Metamodel 40

IRAK Interleukin-1 Receptor-Associated Kinase 1 314

IRAK4 Interleukin-1 Receptor-Associated Kinase 4 . . . 314

ISSI Ingeniería del Software y Sistemas de Información 274

JAR Java Archive 188

JNI Java Native Interface 25

KAOS Knowledge Acquisition in autOmated Specification 351

KEGG Kyoto Encyclopedia of Genes and Genomes . . . 312

KM3 Kernel Meta Meta Model 336

LBP Lipopolysaccharide-Binding Protein 314

LPG LALR Parser Generator 204

LPS Lipopolysaccaride 313

LSC Life Sequence Chart 315

M2M Model-to-model 41

MD2 Myeloid Differentiation protein 2 314

MDA Model Driven Architecture 15

MDD Model Driven Development 31

MDE Model Driven Engineering 4

MDPLE Model Driven Product Line Engineering 4

MDSD	Model Driven Software Development 5
MDSPL	Model Driven Software Product Line 6
MDSPLE	Model Driven Software Product Line Engineering 374
MDT	Model Development Tools 40
MEKK1	Mitogen-activated protein Kinase kinase 1 314
MMDPLE	Multi-Model Driven Product Line Engineering 66
MMDSPL	Multi-Model Driven Software Product Line 7
MMR	Multidimensional Measurement Repository . . . 335
MOF	Meta Object Facility 15
MTF	IBM Model Transformation Framework 41
MVC	Model–View–Controller 38
MyD88	Myeloid Differentiation primary response gene 88 314
NFR	Non-Functional Requirements 351
NF	Nuclear Factor 314
OCL	Object Constraint Language xxiv
OMG	Object Management Group 15
OO	Object-Oriented 74
OSGi	Open Services Gateway initiative 118
OSLO	Open Source Library for OCL 43
PIM	Platform Independent Model 15
PLUSS	Product Line Use case modeling for Systems and Software engineering 55
PSM	Platform Specific Model 15
QVT	Query/View/Transformation xxxiii
RAS	Reusable Asset Specification 220
RDCU	Robotic Device Control Unit 352

RFP	Request for proposal 41
RME	Requirements Model Editor 354
RMME	Requirements Meta-Model Editor 354
RSEB	Reuse-Driven Software Engineering Business . . 54
SBML	Systems Biology Markup Language 313
SBVR	Semantics of Business Vocabulary and Business Rules 40
SDO	Service Data Objects 204
SEI	Software Engineering Institute 68
SME	Scenario Model Editor 358
SMF	Software Measurement Framework 7
SMM	Software Measurement Models 334
SPEM	Software & Systems Process Engineering Meta-Model 84
SPLE	Software Product Line Engineering 6
SPL	Software Product Line 4
SQuaRE	Software product Quality Requirements and Evaluation 351
ST2	Interleukin 1 Receptor-Like 1 315
SWT	Standard Widget Toolkit 182
TCS	Traction Control System 90
TIRAP	Toll-Interleukin 1 Receptor (TIR) domain containing Adaptor Protein 314
TLR4	Toll-Like Receptor 4
TLR	Toll-Like Receptor 313
TRAF6	TNF Receptor-Associated Factor 6 314
UI	User Interface 122
UML2	Unified Modeling Language 2.x 19

UML	Unified Modeling Language 15
URI	Uniform Resource Identifier 36
UTC	United Technologies Corporation 276
W3C	World Wide Web Consortium 40
XMI	XML Metadata Interchange 22
XML	Extensible Markup Language xxxiii
XSD	XML Schema Definition 34
XSLT	Extensible Stylesheet Language Transformations 270

Part I

INTRODUCTION

1 INTRODUCTION

«To feed applied science by starving basic science
is like economising on the foundations of
a building so that it may be built higher.
It is only a matter of time before the whole edifice crumbles.»

— George Hornidge Porter
British Chemist, Nobel Prize in Chemistry in 1967, 1920 – 2002

Software development has become an important industry. Information systems have become increasingly more difficult to develop because of their complex structures, their distributed character, the importance of functional and nonfunctional requirements and their highly dynamic nature. These characteristics have led to that the time spent on software development and maintenance has increased significantly in recent years. Therefore, several proposals and reuse techniques for software development have arisen in order to automate these processes and reduce the time to market.

Since the late '60s the importance of reuse—as a mean to improve the quality and maintainability of the software—was pointed out to reduce development efforts (McIlroy 1968). The reuse of software reduces the implementation time because the same piece of code can be used in different parts of the same application or different applications. Moreover, this mechanism most likely ensures that the

code contains fewer bugs because it has already been used and tested previously (Sommerville 2004).

1.1 MOTIVATION

Reuse mechanisms traditionally have been applied to low-level artifacts (code, modules, etc.). In order to improve these mechanisms in software development processes, the approximation of the Software Product Lines (SPLs) has emerged. This proposal looks forward to implementing industrial development processes for software development (as if an assembly line is involved). It is inspired by the processes to produce physical systems in other areas, e. g. automotive industry, aeronautics, electronics, electrical appliances, etc.

SPLs look forward bringing industrial development processes to the software industry. The goal they aim is to develop not only single software products, but to develop families of products by studying the domain of interest a priori.

A SPL is designed with the aim of developing a family of products, and not only a single and isolated product. This family will consist of products that have a set of common features, and a number of distinct characteristics. In this sense, the reuse mechanisms are designed in an SPL *a priori*; i. e., a domain study of SPL is performed to characterize the entire product family.

A key aspect in this context is the capture and expression of commonalities and variabilities among different products. To reflect the variability of the products, feature models (Kang et al. 1990) are a widespread and accepted notation.

The goal pursued by this thesis is to study the different proposals that have appeared to represent and characterize the variability in recent years; to study their relation with other code reuse and software development paradigms; analyse its advantages and disadvantages and to propose, define, implement and exploit new ideas improving the existing approaches. An important paradigm on Software Engineering is the Model Driven Engineering (MDE), which can provide the foundations needed to implement automated SPL development processes. The merger of MDE and SPL opens a new paradigm of software development which involves a change in the artifacts and processes that are used today: the Model Driven Product Line Engineering (MDPLE). This thesis will focus on the variability view of

MDPLE and how this new conception of SPLs affects to the design of production plans. This thesis makes original proposals, implements them and demonstrates that in some aspects are better than current practices. Finally, it pursues to put the proposal in practise, providing the necessary tools to validate the proposal in different fields of application.

1.2 OBJECTIVES

The Model Driven Software Development (MDSD) community has achieved significant advances by providing tools for the specification and use of models in industrial environments—for example, the Eclipse Modeling Framework (Eclipse 2011a). The SPL community, on their part, has been providing languages and methodologies to deal with the variability in product families, but lacks of the needed standards and tools. The objective of this thesis is to fill the gap between both communities from the variability point of view. Specifically, this thesis aims to:

Both the SPL community and the MDE community can provide great benefits to the software development world. The former, can provide the methodologies to capture variability in a proper way, and the latter can provide the standards and tools to apply such methodologies successfully.

1. Study the feasibility of integrating Software Product Line Engineering and Model Driven Engineering. To consider the possibility of effectively represent the configurations of model features at the instance level (as opposed to the configuration through specialization), to overcome the inability of the current proposals to be integrated with model transformations.

2. Define a metamodel expressive enough to describe the variability in software product families.

3. Implement such a metamodel in an extensible and widely adopted metamodeling tool for industrial environments.

4. Provide mechanisms to define instances of variability models (i. e., configurations) in a simple and user friendly way. These instances must respect the conformance relationships defined by the used modeling standards.

5. Provide the necessary mechanisms to define complex model constraints and the tools needed to check them.
6. Avoid building an inbreeding system. All the previously specified artifacts should be reused and integrated into complex MDE processes.
7. Provide the appropriate graphical editors for all those artifacts, reducing the learning curve using Domain Specific Languages (DSLs).
8. Leverage and reuse the existing industrial tools for automatic code generation, minimizing the development effort and following the MDSD paradigm.
9. Provide an integrated framework to implement complex MDE processes, such as Model Driven Software Product Lines (MDSPLs), and exploit such a framework in different case studies and domains which demonstrate the power of the solution.
10. Demonstrate through a case study that the software artifacts defined by the proposed methodologies and tools are truly interoperable and can be integrated in complex software development processes.
11. Use an example Software Product Line Engineering (SPLE) process to study the implications of the active use of more expressive feature models, and the impact in its development.
12. Validate the scalability of the proposal to represent and deal with large scale models by using a real case study from the industrial area.

1.3 STRUCTURE

This document is structured as follows: part II summarizes the state of the art in MDE and SPLs. Specifically, chapter 2 introduces the main standards on top of which this thesis is built; chapter 3 describes the

main tools that provide support for MDE; and chapter 4 introduces what SPLs are and how variability issues are managed.

Part III studies in depth the variability view and how it can be represented using current metamodeling standards. Chapter 5 introduces the concept of multi-model, and the role the variability view plays in a Multi-Model Driven Software Product Line (MMDSPL). Chapter 6 goes into the main issues which have prevented the use of feature models in complex MDE processes until today; and chapter 7 describes a method to easily and effectively integrate feature models to manage variability in complex MDE processes and more specifically in MMDSPLs.

Next part, part IV, describes the MULTIPLE framework and the main case studies where it has been used to develop and analyse MMDSPLs. Chapter 8 describes in detail the MULTIPLE architecture and user interface. Chapter 9 describes how a traditional SPL is transformed into a MMDSPL and how the latter is implemented using the MULTIPLE framework. Chapter 10 shows how the MULTIPLE framework has been used to represent and analyse a large-scale industrial feature model.

In part V some additional case studies are presented. In these cases, we used the MULTIPLE tool as a framework to implement generic MDE processes. Specifically, chapter 11 shows how MULTIPLE is used to carry out a data transformation process in order to animate and simulate biological processes by using *coloured petri nets*. Chapter 12 presents the Software Measurement Framework (SMF). SMF is a generic framework which uses model transformations to measure any type of software entity. Specifically, SMF uses the MULTIPLE framework to execute such model transformations. Chapter 13 presents MORPHEUS, a tool which provides support for Architecture generaTed from RequIrements applying a Unified Methodology (ATRIUM). ATRIUM is a methodology which allows to define software architectures and requirements concurrently with the goal of generating a proto-architecture of the system's to be. MORPHEUS uses internally the MULTIPLE framework to perform both constraints checking and model transformations.

Finally part VI closes this thesis. In chapter related works are presented and in chapter 15 the conclusions of this thesis are discussed. Chapter 16 presents the different works that have been published throughout the development of this thesis.

Part II

PRELIMINARIES

STATE OF THE ART

SUMMARY

Model Driven Engineering and Software Product Line Engineering are two code reuse techniques which aim to increase the quality of software products and to reduce the time-to-market. Both approaches have gained great relevance in the Software Engineering community; and both have their own languages, standards and tools.

This part of the document describes the foundations of these proposals, which provide the background to this thesis. First, chapter 2 introduces the basic concepts of MDE and its related standards. Second, chapter 3 describes the tools available to implement MDE processes. Finally, chapter 4 describes what SPLs are and which mechanisms they provide to represent and describe the variability of the systems to be.

2 MODEL DRIVEN ENGINEERING: AUTOMATING CODING IN SOFTWARE DEVELOPMENT

«Clearly the most unfortunate people are those who must do the same thing over and over again, every minute, or perhaps twenty to the minute. They deserve the shortest hours and the highest pay.»

— John Kenneth Galbraith
Canadian-American economist, 1908–2006

Technology evolution in the Software Engineering field has made the development of increasingly complex systems possible, specially due to the introduction of techniques that have raised the level of abstraction in the description of problems and their solutions: e. g., structured analysis in the 1970s and '80s (Dahl et al. 1972; Marca and McGowan 1987), object–oriented analysis and design in the '90s (Rumbaugh et al. 1991), etc.

The emergence of Computer Aided Software Engineering (CASE) technology in the 1980s was a big step in this direction. CASE tools aim to provide methods for creating software supporting different analysis and design techniques. These tools allowed developers to express their designs using graphical notations such as structured diagrams or state machines. However CASE technology did not succeed in this decade as expected. The reason must be sought in the

limitations of the translation processes that transferred the graphical representations of software systems (using general-purpose graphical notations) to a specific programming platform or technology.

Advances in the development of programming languages over the past two decades have succeeded in raising the abstraction level in software development, solving the translation problems of the first CASE tools. The advent of languages based on the Object-Oriented paradigm, such as Java, C++ or C#, are more expressive compared to traditional languages like Fortran or C. However, the evolution and maintenance of software systems has become a task that still involves excessive effort.

The advent of the Object-Oriented paradigm has raised the level of abstraction of programming languages, and eases the adoption of MDE techniques. Common standards on MDE are based on object technology, reducing the gap between models and code.

MDE aims at organizing software artifacts at different abstraction levels and support their definition by using software development methodologies, advocating for the use of models as the key artifacts to be built and maintained.

A model consists of a set of elements that provide a precise and abstract description of a system from a view point. The term MDE was proposed by Kent (2002) as a general framework to specify the necessary tasks and models to carry out a software development project entirely.

Any system specification can be expressed using models which may express any aspect of a system. The development process becomes thus a series of refinements and transformations of models where the abstraction level decreases on each step (i. e., models become closer to the implementation platform). In the end, a trivial transformation step is done to generate code because there exists a one-to-one mapping between the most refined model and the code. A MDE process must clearly define the sequence of models to develop at each level and must describe how to refine models in order to decrease the level of abstraction. The system is initially described by means of a model that captures the requirements, regardless of the specifics of the target platform or implementation technology. This is a model with a high level of abstraction which describes only the problem to be addressed.

With the emergence of these technologies, the application of the MDE approaches to the CASE tools only needed to overcome the last obstacle: the lack of standardised modeling languages and methodologies that support the development of software systems throughout the whole project life-cycle. Moreover, the existence of industrial standards is needed to provide real interoperability among different tools.

2.1 MODEL DRIVEN ENGINEERING OPEN STANDARDS

To address the issues presented in the previous section, the Object Management Group (OMG) (OMG 2011b) launched the Model Driven Architecture (MDA) initiative (OMG 2003) as an approach to specify interoperable systems by using formal (or semi-formal) models. In MDA, models that are implementation independent—Platform Independent Models (PIMs) as will be described in the next subsection—are initially expressed in a language which is also implementation independent, such as the Unified Modeling Language (UML). Later, the PIM is translated to another model—the Platform Specific Model (PSM)—which is specific to the desired target platform or language (e. g. Java) by using a set of transformation rules. Finally, starting from the PSM the system code is generated in a object oriented language (Java, C#, . . .). Moreover, the MDA approach proposes to automate the use model transformations and code generation techniques Czarnecki and Eisenecker 2000, thus, the software development process is focused in modeling tasks instead of coding tasks.

The OMG is a consortium founded in 1989 aimed at setting modeling and object-oriented standards. The OMG released their first standard, CORBA (OMG 2004), in 1991. Since then, several specifications which can be considered as a de facto *standards in industry are promoted by them, (e. g. UML or MOF).*

MDA relies on a great amount of the OMG standards, some of them are the following:

META OBJECT FACILITY is the common language (meta-meta-model) used to describe metamodels in the MDA approach. Subsequent metamodels (such as UML) are described by using Meta Object Facility (MOF).

UNIFIED MODELING LANGUAGE provides a language to describe different systems. UML is a domain-independent language, although its origins are in the object-oriented modeling.

This thesis aims to use as more standardised specifications as possible in order to be generic and interoperable. This way, the OMG standards are a fundamental basis for the understanding of this document.

OBJECT CONSTRAINT LANGUAGE is a declarative language without side-effects to express queries and describe constraints over MOF and UML models.

QUERY/VIEW/TRANSFORMATION is a standard to describe model transformations and equivalence relationships among MOF-based models. It uses OCL to express complex queries over the candidate models (i. e., the models that take part in a given transformation).

XML METADATA INTERCHANGE is an XML-based language that provides the persistence mechanisms to store and interchange models among MOF-compatible tools.

2.2 MODEL DRIVEN ARCHITECTURE

MDA is a software design approach whose purpose is to produce executable systems. MDA is focused specially in changing technology, integration and interoperability problems.

As discussed before, the OMG has proposed a MDE framework known as MDA which aims to establish itself as a *de facto* standard. MDA's main goal is to solve the problem of changing technology and integration. The main idea behind it is to use models, so that the system properties and features are reflected in an abstract description. Thus, models are not affected by technological changes.

MDA includes a software development process, therefore, its purpose is to produce executable software systems. Moreover, MDA expects to solve some problems such as low productivity in software development and interoperability issues (Kleppe et al. 2003) . The former is solved performing the analysis and development of systems by evolving high-level models from which there would be possible to automatically generate code. Interoperability can be solved because code generation techniques allow to obtain code for different technologies, which adds another advantage: the reuse of models.

Some basic concepts defined in the MDA standard are—quoted from OMG (2003):

PLATFORM — *"A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented."*

PLATFORM MODEL — *"A platform model provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided by that platform. It also provides, for use in a platform specific model, concepts representing the different kinds of elements to be used in specifying the use of the platform by an application."*

PLATFORM INDEPENDENT MODEL — *"A platform independent model is a view of a system from the platform independent viewpoint. A PIM exhibits a specified degree of platform independence so as to be suitable for use with a number of different platforms of similar type."*

PLATFORM SPECIFIC MODEL — *"A platform specific model is a view of a system from the platform specific viewpoint. A PSM combines the specifications in the PIM with the details that specify how that system uses a particular type of platform."*

Development of systems in MDA begins with the construction of a PIM. A PIM is transformed to one or more PSM and finally, the code is generated from the PSM. The fundamental operation in MDA is to transform PIMs into PSMs.

The development of a system according to the MDA framework begins with the construction of a PIM. Later, that PIM is transformed to one or more PSM. Finally, the code is generated from the PSM.

The fundamental operation in MDA is to transform PIMs into PSMs. The MDA guide (OMG 2003) states that PIMs and PSMs models are

expressed as UMLs models and transformations must be automated as much as possible.

Mellor et al. (2004) classify transformations into two types: vertical and horizontal. A transformation is horizontal if the source model and target model belong to the same level of abstraction. In MDA this can be a PIM to PIM transformation or a PSM to PSM transformation. A transformation is vertical when the source model and target model are located at different levels of abstraction. In MDA, transformations from PIMs to PSMs or PSMs to code are vertical transformations. From the generic point of view of MDE, transformations from PIM to PIM and from PSM to PSM are also interesting (according to the classification of models defined by MDA) as can be used to refine models.

Initial versions of MDA served as the background for MDE, which generalizes the concepts of MDA, and defines the transformations in the context of metamodeling. In MDE, the most accepted way to define the models is through metamodeling techniques, and transformations are defined using model transformation languages. In contrast, in the original proposal of MDA, metamodeling was not a necessary condition. It was only in later versions of MDA where the ideas defined by MDE were incorporated, which led to the definition of the MOF and QVT standards.

2.3 META OBJECT FACILITY

MOF is the basic vocabulary used in MDA to define metamodels. A metamodel can be considered as the abstract syntax of a language, and in fact, MOF can be used to define new DSLs.

MOF is the basic vocabulary (called meta-metamodel) provided by the MDA standard to describe metamodels, and therefore new vocabularies (in fact we could say *abstract languages*, but we use the term metamodel to avoid confusion). We can define new metamodels using exactly the same tools used to define models.

However, one may wonder if there is a superior model vocabulary used to define meta-metamodels. The answer is yes, this model of meta-metamodels would be called meta-meta-metamodel. But, as this artifact is also a model, could we continue to expand this pyramid endlessly?

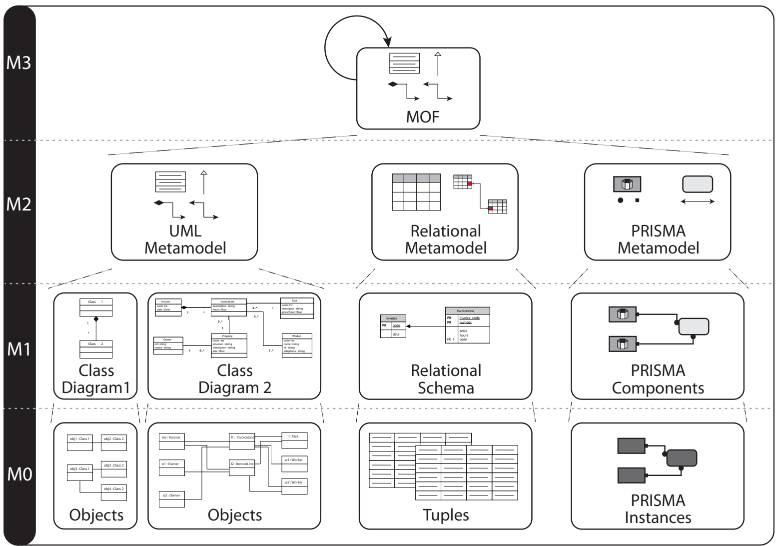


Figure 2.1: MOF layers

In practice this makes no sense, and models and metamodels are often organized into a four-layer M3-M0 structure with the following distribution:

- The level M3, which closes the structure at the top, contains the basic vocabulary to describe metamodels. It should be noted that this level usually contains a unique vocabulary (meta-metamodel) that characterizes the modeling approach chosen. This meta-metamodel must be defined using its own constructs, thus, the structure is closed in this layer.

  In MOF, the meta-metamodel is MOF itself. The syntax of the MOF language can be considered as a subset of the Unified Modeling Language 2.x (UML2) class diagram.

- Metamodels, the languages to describe new models, are found in the lower layer, the M2 layer. MOF–compliant metamodels are defined in terms of *classes*, *attributes*, *associations*, etc., which are the constructs provided by the meta-metamodel.

- Models are placed at the M1 layer. As we have introduced earlier in this thesis, a model is an abstract description of a system.
- The lower level, the M0 level, is were data are found, i. e., the instances of the system under study.

This four-layer structure (shown in Fig. 2.1) can get a wealth of vocabulary to describe different types of systems, or provide different views of the same system.

It is noteworthy to remark that this fixed hierarchy of levels can be sometimes confusing. Perhaps, it is more interesting to look directly to the relationship between a model and vocabulary, and realize that this relationship occurs at all the levels described. This relationship is usually called "*instance-of* relationship". We say that a model x is an instance of a vocabulary $x + 1$, which is called metamodel. The model is on the lower level (the instance level), and the metamodel in the upper level (meta level). We can apply this duality to the metamodel $x + 1$, and if we consider $x + 1$ as the instance level, we see that also $x + 1$, necessarily, is defined by a vocabulary $x + 2$. Therefore we can put a model at both the meta level and say that it has instances, or at the instance level, and say that it is defined using a metamodel. It is worth noting the special case of the meta-metamodel (level M3) that defines itself, so we could say that is an instance of itself.

OCL is a language closely related to UML. Initially, it was designed to describe constraints over the elements of UML models in order to specify systems with a great level of detail. With new versions of UML, OCL was completed. Nowadays it is possible to add almost any expression to an element of a UML diagram.

MDA places at the M2 layer several well-known metamodels which are defined using MOF, such as UML (OMG 2010b), OCL (OMG 2010a) or QVT (OMG 2008a).

2.4 OBJECT CONSTRAINT LANGUAGE

OCL is a notational language for the analysis and design of software systems. It is defined as a standard language to complement UML. Specifically, OCL gives support to UML for specifying constraints and queries on models, allowing to define and document UML models more precisely.

A UML model, such as a class diagram or state diagram, does not constitute a sufficiently precise and unambiguous specification of a system. OCL expressions complete this specification, providing additional information for object-oriented models and other modeling assets. Usually, this additional information can not be expressed by means of a diagram.

Each OCL expression refers to a type (e. g. *class*, *interface*, ...) defined in a UML diagram. Thus, an OCL expression is always linked to a UML diagram and it does not make sense in isolation.

2.4.1 *Language features*

In UML 1.1, OCL was introduced as a language to express constraints over the elements of a model, defined as restrictions over the attribute values and instances of an object-oriented model or system.

UML is designed to specify software systems. However, many models are not directly executable in early stages. Using OCL it should be possible to verify the correctness of these systems by checking their OCL expressions without having to produce an executable version of the models.

In UML 2.x additional constructs were included to define queries, reference values, status conditions, business rules, etc. In short, OCL can be used to associate any expression on elements of a diagram.

OCL expressions can appear anywhere in a model to indicate a *value*. A *value* can be a single value such as an integer, but may also refer to an object, a collection of values, or a collection of object references. An OCL expression may represent, for example, a boolean value used in the condition of a state diagram, or a message in an interaction diagram. An OCL expression may refer to a specific object in an interaction or and object diagram.

OCL is based on set theory and predicate logic, and has formal mathematical semantics (Richters and Gogolla 2000). Its notation, however, does not use mathematical symbols. Thus, OCL provides the rigor and precision of a formal language and an ease of use close to that of the natural language.

OCL expressions are used to model and specify systems. However, many models are not directly executable, and many will contain OCL expressions even if no executable versions of the system exist. However, it should be possible to verify the correctness of these expressions, without having to produce an executable version of the

model. Since OCL is a typed language its expressions can be checked during modeling before implementation. Thus, model errors can be eliminated early.

Another essential aspect is that OCL is a declarative language. In procedural languages, like most programming languages, statements are descriptions of the actions that need to be carried out. In a declarative language, an expression describes *what* should be done, but not *how* to do it. To ensure this, OCL expressions have no side effects, i. e., an OCL expression can not change the system's state.

2.5 QUERY/VIEW/TRANSFORMATION

QVT is the language proposed by the OMG to describe and perform model transformations. Model transformations defined in the remaining of this thesis will be described using this standard.

Model transformations are a key issue in MDE. They guide the software development process and allow to derive implementation code from PIMs and PSMs. With the aim of providing a suitable framework for model transformations, the OMG has proposed the Query/View/-Transformation (QVT) standard (OMG 2008a) to complete the MDA proposal. QVT lays on other two OMG standards, namely MOF 2.0 and OCL 2.0, reusing previous technology and reducing the learning curve of the implementation.

The QVT specification is defined by two orthogonal dimensions: *language* and *interoperability*, each of which has a number of levels. The intersection of two dimensional levels defines a *QVT conformance point*.

The language dimension defines the different transformation languages that the QVT specification defines. Specifically there are three: *Core*, *Operational* and *Relations*, being the main difference among them their declarative or imperative nature.

In the interoperability dimension we find those features that allow a compliant tool interoperate with other tools. Specifically, there are four interoperability levels, namely *syntax executable*, *XML Metadata Interchange (XMI) executable*, *syntax exportable* and *XMI exportable*, according to the standard specification (OMG 2008a):

SYNTAXEXECUTABLE — *"An implementation shall provide a facility to import or read, and then execute the concrete syntax description of a transformation in the language given by the language dimension. The execution shall be according to the semantics of the chosen language [...]."*

XMIEXECUTABLE — *"An implementation shall provide a facility to import or read, and then execute an XMI serialization of a transformation description that conforms to the MOF meta-model of the language given by the language dimension. The execution shall be according to the semantics of the chosen language [...]."*

SYNTAXEXPORTABLE — *"An implementation shall provide a facility to export a model-to-model transformation in the concrete syntax of the language given by the language dimension."*

XMIEXPORTABLE — *"An implementation shall provide a facility to export a model-to-model transformation into its XMI serialization that conforms to the MOF meta-model of the language given by the language dimension."*

2.5.1 *Languages*

Of the different languages QVT provides, the Relations *language is preferred. This thesis uses this declarative language, as it provides a graphical notation that is very intuitive and easy to understand.*

The QVT specification has a hybrid declarative/imperative nature, with the former divided into a two-tier architecture that conforms the basic framework for the execution semantics of the imperative part.

2.5.1.1 *Declarative languages*

The two declarative layers are structured as follows:

It is possible to draw an analogy between the declarative layers of QVT and the Java™ architecture, where the Core *language is like the* Java Byte Code *and the semantics of* Core *behave like the Java virtual machine. The* Relations *language plays the role of the Java language, and the transformation from a* Relations *specification to a Core specification is comparable to the specification of a compiler that generates* Java Byte Code.

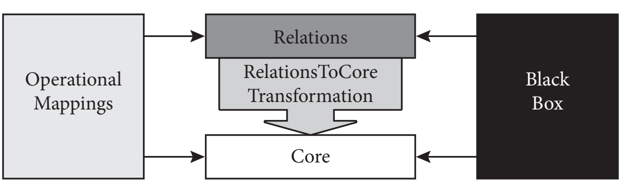


Figure 2.2: Relationships among the QVT metamodels

A *RELATIONS* METAMODEL (and a user-friendly language that gives both graphical and textual support for it) which enables the definition of pattern-matching expressions (*object templates*) over complex sets of objects. The traceability links among the models involved in a *Relations* transformation are automatically created and maintained.

A *CORE* METAMODEL (and its associated textual language) which is defined as a minimal superset of the Essential Meta-Object Facility (EMOF) and OCL standards which provide basic transformations capabilities. All the traceability classes must be explicitly defined as EMOF models, and the maintenance of the traceability links must be done manually as any other regular object involved in a transformation.

2.5.1.2 *The* Operational Mappings *language*

This language is specified as the standard way to provide imperative implementations of QVT transformations. It provides some extensions to OCL which provide it with a more procedural style, and a syntax closer to imperative languages. It is possible to write complete model transformations using the operational mappings language. Such transformations are called *operational transformations*.

2.5.1.3 *Black Box Implementations*

The QVT standard allows the definition of *black box* implementations, in order to *plug-in* any MOF Operation with the same signature than a Relation. This is useful for several reasons:

If we extend the Java analogy, the ability to invoke black-box and operational mappings implementations, can be considered equivalent to using the JNI .

- It allows to implement complex algorithms in any programming language (provided that it has a MOF binding).
- It allows the use of domain specific libraries to calculate certain model properties.
- It allows to hide the implementation of certain parts of a model transformation.

However, this integration can be dangerous since the black-box implementations have free access to the object references of the models, breaking some encapsulation mechanisms. Moreover, black-box implementations do not have an implicit mapping among the domains they transform, so that every black-box must explicitly maintain the traceability links among the candidate objects of the transformation (as a relation does automatically).

2.5.2 *The* Relations *language*

Next, the main features of the *Relations* language are explained. This language will be extensively used throughout this document.

2.5.2.1 *Transformations and Model Types*

In the *Relations* language, a transformation among a set of candidate models is specified as a set of relations that must hold for the transformation to be successful. A candidate model is any model that conforms to a *model type* (metamodel). Candidate models have a name, and the types of the items that they may contain are restricted to the elements of the metamodel they conform to.

TRANSFORMATION EXECUTION DIRECTION A transformation invoked for enforcement, i. e. with the goal of effectively modify a candidate model, must be executed in a particular direction by selecting one of the candidate models as the target (destination) model. The target model may be empty, or may contain elements that will be related by the transformation. The transformation proceeds as follows: first, it checks if all the relationships and constraints defined in the transformations are met, and, second, when a relationship does not hold, the target candidate model is modified (by creating, modifying or deleting elements) in order to make the relationship hold.

2.5.2.2 *Relations and domains*

Relations in a transformation define constraints that must be satisfied by the elements of the candidate models. A relation defined for two or more *domains*, and a pair of pre- and post-conditions (*when* and *where* predicates as explained next), specify a relationship that must be satisfied by the elements of the candidate models.

A *domain* is a typed variable to be matched in a model of a given metamodel. A domain defines a pattern that can be viewed as a graph of object nodes, their properties and the association links originating from an instance of the type of domain. Alternatively, a pattern may be considered as a set of variables and a set of constraints that model elements bound to those variables must satisfy to validate that pattern.

When and *where clauses can be considered as the pre- and post-conditions that apply to a given relation respectively.*

WHEN AND WHERE CLAUSES A relation may be constrained by two sets of predicates: the *when* and *where* clauses. The *when* clause defines the conditions under which a relation needs to hold. The *where* clause defines the condition that must be satisfied by all the elements of the models participating in the relationship, and may restrict any of the variables of the relation and its domains.

When and *where* clauses may also contain any OCL expression besides any relation invocation. Relation invocation allows us to build complex relations based on simple relations.

TOP-LEVEL RELATIONS A transformation may contain two types of relations: *top-level* and *non-top-level*. The execution of a transformation requires that all its *top-level* relations hold, while *non-top-level* relations must hold only when they are invoked directly or transitively from the *where* clause of another relationship.

Top-level relations are directly executed by the QVT–Relations transformations engine. If a non-top level relation is not specified as a pre- or post-condition of another relation it will not be executed.

A *top-level* relation is syntactically distinguished from a *non-top-level* relation by the *top* keyword.

CHECK AND ENFORCE Whether or not the relationship may be enforced (i. e., it performs a model transformation) is determined by the destination domain, which can be marked as *checkonly* or *enforce*. When a transformation is executed in the direction of a *checkonly* domain, it simply checks if a valid correspondence of the relevant model satisfies the relationship. When a transformation is executed in the direction of a *enforce* domain, if the test fails, the target model is modified as necessary to satisfy the relationship.

2.5.2.3 Pattern-Matching

The patterns which are contained in the domains of a relation are known as *object template expressions*. A relation may define different *object template expressions* which will be used in the pattern matching of the candidate models.

A template expression match results in a binding between the model elements and the variables declared in the domain. A match may be performed in a context where some variables have been already bound to a model element (for example, in a *when* clause or other object template expression). In this case, the match only binds the free variables.

Arbitrarily deep nestings of template expressions are permitted, and matching and variable binding proceeds recursively until there

is a set of value tuples corresponding to the variables of the domain and its template expression.

2.5.2.4 *Keys and object creation using patterns*

Keys *are a fundamental part of the QVT–Relations language. The use of keys ensures that objects will not be duplicated. An object will not be created if there already exists another object with the same values in its key attributes.*

As mentioned previously, an *object template expression* also serves as a template for creating an object in a target model. When for a given valid match of the source domain pattern, there does not exist a valid match of the target domain pattern, then the object template expressions of the target domain are used as templates to create objects in the target model.

However,when creating objects we need to ensure that duplicate objects are not created when the required objects already exist. In such cases we just want to update the existing objects. But how do we ensure this? The MOF allows for a single property of a class to be nominated as identifying. However, for most metamodels, this is insufficient to uniquely identify objects. The relations metamodel introduces the concept of *Key*, which defines a set of properties of a class that uniquely identify an object instance of the class in a model. A class may have multiple keys (as in relational databases).

Keys are used at the time of object creation; if an object template expression has properties corresponding to a key of the associated class, then the key is used to locate a matching object in the model; a new object is created only when a matching object does not exist.

2.5.2.5 *Executing a transformation in* checkonly *mode*

A transformation can be executed in *checkonly* mode. In this mode, the transformation simply checks if the relations are satisfied in all directions, and reports errors when the relations do not hold. No enforcement is done in any direction, regardless how the domains are marked (*checkonly* or *enforce*).

2.6 SUMMARY

We have presented an overview of the most important MDE standards proposed by the OMG and they are included in order to make this thesis a self-contained document. The main goal of this chapter was to give a concise introduction to MDA an the most important standards and languages related to it, i. e., MOF to describe metamodels, models and instances; OCL to define rich model constraints; and QVT to describe correspondences and model transformations among MOF models. These are the basic standards on top of which this thesis is built.

3 SUPPORTING TECHNOLOGIES FOR MDE

«La técnica es el esfuerzo para ahorrar esfuerzo (Technology is the effort to save effort)»

— José Ortega y Gasset
Spanish philosopher and humanist, 1883–1955

The Model Driven Development (MDD) trend (Selic 2003) is aligned with the MDE principles. MDD considers models as the main assets in the software development process. Models collect the properties that describe the information system at a high abstraction level, which permits the development of the application in an automated way following generative programming techniques. In this process, models constitute software artifacts that experience refinements from the problem space (where they capture the requirements of the application) to the solution space (where they specify the design, development and deployment of the final software product).

In this context it is essential to have the proper tool support to define new models and metamodels, and to provide the necessary implementations of the standards and languages proposed by the industrial consortia, such as the OMG. Without the adequate tools the new paradigm of MDD can not be implemented in any software development process. Traditionally, the tools that provided support

for modeling tasks were either context-specific, i. e., metamodel-specific (ontologies, relations databases, etc.) or technology-specific. With the advent of MDA and MOF, new modeling tools are being designed to be easily customizable, providing support to define custom models and metamodels. The adoption of MOF is also increasing the importance of the DSLs trend. Thus, such modeling tools are being designed following a new paradigm where modularity and extendability are key attributes to maximize. In this context, the release of the Eclipse platform, and its wide ecosystem of satellite projects, represents a turning point in the tooling support for the MDE trend and all its associated standards (MDA, MOF, QVT, OCL, etc.). Throughout the following section we will present the Eclipse platform and the most relevant projects and tools that provide support to the different standards that are employed in the implementation of this thesis.

3.1 THE ECLIPSE PLATFORM

Eclipse is a tool that provides a generic runtime which implements a complete and dynamic component model. This component model allows developing any kind of software application in a modular way. Tools built in this thesis are built on top of Eclipse thanks to this runtime.

Eclipse is an open source software development environment, whose purpose is to provide a platform for highly integrated tools. Eclipse consists of a central project that includes a generic core framework for integrating tools, and a Java development environment built using the previous framework. Other projects extend the core framework to support other types of tools and specific development environments. Eclipse projects are implemented in Java and run on different operating systems, including *Windows*, *Mac OS X* and *Linux*.

The *Eclipse Foundation*, founded in 2004, is a not-for-profit foundation of several companies that have committed to provide support to the Eclipse project in terms of time, expertise, technology or knowledge. This organization was created by *Eclipse.org*, a consortium founded in 2001 when IBM released the Eclipse Platform into Open Source.

The Eclipse Platform is a framework to build Integrated Development Environments (IDEs). It is usually described as *an IDE for everything and nothing in particular* (Eclipse 2003), as it simply defines a basic IDE structure. Specific tools extend this basic framework,

and are plugged-in to define a particular IDE configuration all together.

The basic unit of functionality (e. g., a component), is called a *plug-in*. The Eclipse platform itself, and the tools that extend its functionality are made of plug-ins. A single tool can be a single plug-in, but more complex tools are typically divided into several plug-ins.

From a packaging perspective, a plug-in includes everything needed to run a component, such as Java code, images, localized text, etc. It also includes some manifest files (`MANIFEST.MF` and `plugin.xml` typically), which declare the connections with other plug-ins. The manifest declares, among other things, the following items:

REQUIRES — the dependencies from other Eclipse plug-ins.

EXPORTS — which internal classes will be visible for other Eclipse plug-ins.

EXTENSION POINTS — declaration of functionality points (defined at a high level of abstraction) which are public. Other plug-ins may connect to these points.

EXTENSIONS — which extension points are used. These extension points may be declared by other plug-ins.

When Eclipse is launched, it scans and discovers all the installed plug-ins and connects the *extensions* with the corresponding *extension points*. In the following sections we describe the main frameworks of the Eclipse ecosystem which are related with modeling and metamodeling tasks: the Eclipse Modeling Framework (EMF) and the Graphical Modeling Framework (GMF).

3.1.1 *Eclipse Modeling Framework*

The Eclipse Modeling Framework (EMF) provides modeling, metamodeling and code generation capabilities within the Eclipse plat-

form. Additionally, it can be used as a standalone library to deal with models and metamodels in Java applications.

EMF is the modeling and metamodeling framework provided by Eclipse. Its modeling language, called Ecore*, can be considered as an implementation of MOF.*

EMF is just an environment to describe models and their instances, and is used to generate new software artifacts from a model description (such as a Java implementation of the model). EMF moves on the direction of MDA, but it is not fully aligned with it since it has been designed from a practical point of view, keeping in mind implementation details of Java programs.

EMF allows to define models in different ways. Traditionally, models were built using annotated Java, XML Schema Definition (XSD) or UML models from Rational Rose. Nowadays, it is quite common to use EMF-based class diagrams or UML models from the Eclipse UML2 project. The capabilities of the framework remain the same regardless of the way used to define the EMF model. EMF uses *Ecore* (Steinberg et al. 2009) as the canonical language to describe models, and thus, any of the previous ways to define an EMF model generates an *Ecore* model in the end.

3.1.1.1 *EMF models*

An *Ecore* model is, essentially, a subset of the UML class diagram and thus, can be considered as an implementation of the EMOF language proposed by the OMG (OMG 2006). This way, an *Ecore* model is a model of the classes of a software application (i. e., the structural description). Because of this, several benefits of modeling can be obtained in an standard Java development environment, given that the correspondence between an *Ecore* model and its Java implementation is natural and straightforward.

An *Ecore* model describes, however, concepts at a higher level of abstraction than mere classes and attributes. Looking at the generated Java code that implements an *Ecore* model we will note that, for example, attributes are transformed into *getter* and *setter* methods enforcing encapsulation. Unlike traditional implementation, these

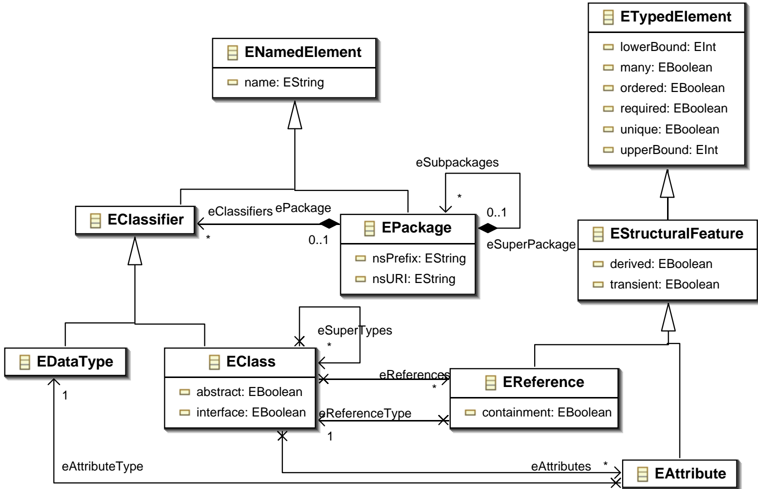


Figure 3.1: Simplified version of the *Ecore* metamodel

methods have the ability to notify changes to different *observers*[1] (Gamma et al. 1995). In addition, references can be bidirectional, and referential integrity is always maintained automatically. These references can also exist among different resources (documents), etc.

3.1.1.2 *The* Ecore *(meta)model*

As noted before, the metamodel used to describe EMF-based models is *Ecore*. Following an approach similar to MOF, *Ecore* is defined using *Ecore* itself, which implies that *Ecore* is the meta-metamodel of the *Ecore* metamodel.

Ecore is a language designed to define any kind of metamodel. With this goal, it provides the necessary constructs to describe concepts and the relationships among these concepts. Using *Ecore* we can define new vocabularies (or DSLs) which allow us to work with

1 The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes.

models in different contexts. Fig. 3.1 shows a simplified subset of the *Ecore* metamodel.

The main element is *EClass*, which models the concept of class. Its semantics is similar to that of UML *Classes*. *EClass* is used to describe new concepts in *Ecore*. An *EClass* comprises a set of attributes (*EAttributes*) and a set of references (*EReferences*) to other *EClasses*, as well as any number of superclasses (also as the UML standard does). The remaining elements of the *Ecore* metamodel are the following:

ECLASSIFIER is the abstract datatype which groups all the elements that describe concepts.

EDATATYPE is used to represent the type of an attribute. An *EDataType* can be a basic datatype (such as `integer` or `float`), an *EClass*, or a java class (such as `java.util.Date`).

EATTRIBUTE is the element used to define the attributes of an *EClass*. *EAttribute* has (among other properties) a name and a type. As a specialization of *ETypedElement*, *EAttribute* inherits some properties such as cardinalities (*lowerBound* and *upperBound*), if it is required or not, if it is a derived attribute, etc.

EREFERENCE allows to model relationships among *EClasses*. Specifically, an *EReference* can be used to model associations and compositions as the UML standard does. As well as *EAttribute*, *EReference* is an specialization of *ETypedElement* and inherits the same properties.

EReference also has the *containment* property, which is used to model the disjoint aggregations (or compositions as described in UML).

EPACKAGE is used to group a set of *EClasses* in a modular way as UML packages do. Its main attributes are the name, the prefix (*nsPrefix*) and the Uniform Resource Identifier (URI). The URI is a unique identifier used to reference the *EPackage* unambiguously.

Similarities between EMF and UML are evident, given that MOF is based in UML class diagrams. However, *Ecore* is preferred by the Eclipse community over UML it is is smaller and simpler. But unlike UML, it can not model the behaviour of a system in addition to its structure. Another particularity that we find in the context of EMF, is that an *Ecore* model is only composed by a single root *EPackage*. This way, an *Ecore* model is identified by the URI of its unique root *EPackage*.

3.1.1.3 *XMI serialization*

As described previously, a *conceptual* model can be described in EMF in different ways (Java code, XSD, Rational Rose, etc.), however, the canonical representation of a conceptual model is *Ecore* and its persistence format is XMI (OMG 2011c), the standard proposed by the OMG for MOF metadata exchange.

XMI is the default persistence format used in EMF. It is used to store both models and instances.

Both Eclipse and EMF provide automatic support for XMI persistence for *Ecore* model. Moreover, when an *Ecore* model is used to create the Java implementation of a given system, it automatically creates the code used to retrieve and store the model instances in secondary storage (i. e., the program data themselves).

3.1.1.4 *Code generation*

The main advantage of EMF (which is common to other modeling frameworks and techniques) is the increase in the productivity brought by the code generation mechanisms. From an *Ecore* model it is possible to obtain a Java implementation only with a few clicks by using EMF built in wizards. Moreover, as EMF is generic, it can be used to build new code projectors to different technologies. This can be used to provide textual representation to any arbitrary DSL, or can be used to generate full implementations in any Object-Oriented language. An example of the latter case is the *EMF4CPP* project (González et al. 2010). This project is able to generate C++ code form an *Ecore* model, among other features. Nevertheless, this project is not only a C++ code generator, it also reimplements some

of the EMF functionality in C++ which enables the execution of the generated C++ programs.

3.1.2 *Graphical Modeling Framework*

GMF is a framework which bridges a drawing framework and a modeling framework to automatically build graphical editors for DSLs. Graphical editors presented in this thesis are built following a MDD approach by using GMF.

The Graphical Modeling Framework (GMF) (Eclipse 2011e) emerges to cover a need in the development of Graphical User Interfaces (GUIs) and graphical editors in Eclipse. Traditionally, the development of graphical editors in Eclipse has been done by using the Graphical Editing Framework (GEF). GEF is a library that uses the Model–View–Controller (MVC) architectural pattern (Burbeck 1987), and is composed by several plug-ins. Specifically, the `org.eclipse.draw2d` plug-in provides the basic functionality for rendering and ordering graphical elements in a canvas.

As GEF uses the MVC pattern, developers had to define their own models, and had to build the corresponding mechanisms to store an retrieve these models. Given that EMF provides a generic framework for modeling, and it provides the persistence mechanisms for free, it is the perfect tool to build these models in an easy way. This way, to easily build new graphical editors we only need to map these model elements (provided by EMF) with the graphical elements (provided by GEF and `org.eclipse.draw2d`). Following a MDD approach, this mapping is be done by using models. Thus, GMF is a bridge between EMF and GEF to build DSLs editors.

Figure 3.2 shows the main components and models used during GMF-based development. The process begins with 3 models: the *domain model*, the *graphical definition* and the *tooling definition*. The first one corresponds to the EMF model for which we wish to create the new editor. The second one describes which graphical primitives will be drawn in the editor without defining any correspondence with the domain model elements. The third model defines the tools that are shown on the palette of the editor (and other GUI elements such as menus, toolbars, etc.). These tools are used to create new elements and to draw links among them.

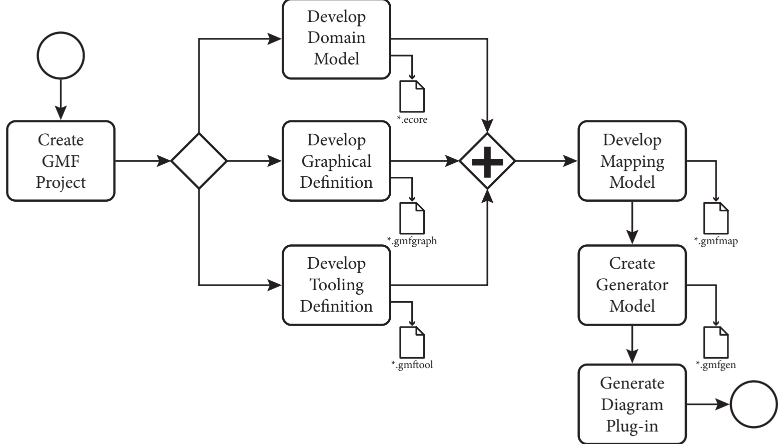


Figure 3.2: GMF workflow overview

The use of different models in GMF favours modularization and reuse. Graphical definitions are separated from the domain elements they represent.

A graphical definition can be valid for different domains. For example, in the UML class diagram we find different elements that are extremely similar in appearance and structure. In GMF a graphical definition can be reused for different domains. This is achieved by using a separate model called *mapping model* which links the graphical elements and the tool definitions with the desired elements of the domain model.

Once the mappings have been defined, GMF provides a *generator model* that can be used to tune the last implementations details used in the automatic code generation phase. The production of an editor plug-in based on the *generator model* will target a final model, the diagram runtime or *notation model*. The runtime will bridge the notation and domain model when a user is working with a diagram, providing their persistence and synchronization.

3.1.3 *Model Development Tools*

The Eclipse Model Development Tools project aims to provide an EMF-based implementation for several modeling standards, most of them promoted by the OMG.

The Model Development Tools (MDT) project (Eclipse 2011b) is an Eclipse project which aims to provide both a set of industry standard metamodels and the tools to work with them. Its creation is relatively recent, and emerges as a reorganization of different small modeling projects inside the Eclipse ecosystem. Some of the subprojects that this project hosts are: *BPMN2*, implementing the next Business Process Model and Notation (BPMN) v.2 (OMG 2011a) standard; *IMM*, which implements the forthcoming Information Management Metamodel (IMM) (OMG 2005a) specification; *MoDisco* (Bruneliere et al. 2010), used for software modernization; *MST*, a subproject to give full compatibility with Complete Meta-Object Facility (CMOF) (OMG 2006) specifications; *Papyrus* (Eclipse 2011c); and *SVBR*, which implements the Semantics of Business Vocabulary and Business Rules (SBVR) specification (OMG 2008b). Other metamodels and tools that compose the MDT project which are used in the implementation of this thesis, and are key components of this work are:

OCL subproject provides an implementation of OCL for EMF-based models and metamodels. This set of utility libraries are used and integrated in different tools that have been produced in the implementation of this thesis (see 8.5.1 and 8.5.2).

UML2 is the reference project, which implements the UML2 metamodel using *Ecore* as its meta-metamodel.

UML2 TOOLS provide a set of GMF-based editors for drawing and editing UML2 models. It aims to provide automatic generation of editors for all UML diagram types.

XSD is the library which implements the XSD metamodel, and provides an Application Programming Interface (API) for manipulating its instances as described by the World Wide Web Consortium (W3C) XSD specifications (Gao et al. 2008), as

well as an API for manipulating the Document Object Model (DOM)-accessible representation of XML.

3.2 MOMENT: A FRAMEWORK FOR MODEL MANAGEMENT

MOMENT (Boronat et al. 2005b; Boronat 2007) is a tool that provides support to different OMG standards including capabilities to transform models. The tool uses both an industrial modeling front-end and an algebraic back-end for the execution of the transformation and query tasks. The algebraic background runs in the high performance rewriting system called Maude (Clavel et al. 2002). The industrial modeling environment used by MOMENT is Eclipse and EMF.

In its first version, MOMENT used the QVT standard to provide a transformations language, unlike other popular transformation tools, such as IBM Model Transformation Framework (MTF) (Demathieu et al. 2005) or ATLAS Transformation Language (ATL) (INRIA 2011), which provide their own proprietary languages. The tool offers an implementation of the *QVT Relations* language as well as the OCL language. For this language, MOMENT gives wide support for unidirectional transformations. Moreover, the tools provides full support to the query operators of the OCL language.

3.3 ATLAS TRANSFORMATION LANGUAGE

ATLAS Transformation Language (ATL) is a transformation language and a toolkit initially by the *AtlanMod team*—initially *ATLAS Group*—(AtlanMod 2011). ATL is a Model-to-model (M2M) transformation language which was designed as an answer to the OMG MOF/QVT Request for proposal (RFP) (OMG 2002)—cf. ATLAS (2005); Jouault and Kurtev (2006). The ATL proposal was rejected in favor of the proposal made by Appukuttan et al. (2003). Nevertheless, ATL has gained relative popularity and has become an Eclipse project (INRIA 2011) inside the MDT project (Eclipse 2011b). Nowa-

days ATL is published under the terms of the open-source Eclipse Public License (EPL).

Although ATL is aligned with the QVT standard, it does not provide the same architecture nor a pure declarative transformation language; rather it provides a hybrid language (imperative/declarative) to perform MOF-compliant model transformations.

3.4 IBM MODEL TRANSFORMATION FRAMEWORK

IBM Model Transformation Framework (MTF) (Demathieu et al. 2005) was the proposal of IBM in order to help with the standardisation of QVT. MTF provided a prototype which used EMF as it underlying modeling framework. A model transformation was defined in MTF by defining relationships among model elements using, for example, the *relate* and *equals* keywords. Although this proposal is quite similar to the QVT–Relations language, MTF did not succeed and the tool remained as a prototype. Nowadays MTF has been abandoned in favor of other languages, e. g. QVT and ATL, and the MTF site has been shut down.

3.5 MEDINI QVT

mediniQVT is a tool entirely implemented in Java, and integrated as a set of Eclipse plug-ins. This tool supports model transformations using the QVT standard. It has been developed by the company *ikv++ technologies ag* (ikv++ 2011), located in Germany. The engine was originally released as a free product in 2007—*free as in free beer* (FSF 2011)—and was in early 2008 when the source code was released under the EPL open source license.

From the technical point of view, *medini QVT* uses EMF as its modeling and metamodeling environment. For the definition of model transformations implements a *QVT Relations* engine, giving support for complex model-to-model transformations. Internally,

medini QVT uses the Open Source Library for OCL (OSLO) toolkit (OSLO 2011).

The main features of the tool are:

- Allows QVT transformations using the textual concrete syntax of the *QVT Relations* language.
- Provides a textual editor with syntax highlight and code completion capabilities.
- Includes an advanced debugger to trace the execution of model transformations step by step throughout the application of the different relations.
- Implements the concept of *key* of the *Relations* language, allowing the execution of incremental transformations.
- Supports transformations with more than two different domains (*n-domain* transformations).
- Is able to execute bidirectional transformations (if the transformation is unambiguous in its definition).

mediniQVT *is the most complete and user-friendly tool which provides support to QVT–Relations. Moreover, its core transformations engine is available under an open source license. For this reason, parts of this engine are reused in the tools developed in this thesis.*

All this functionality is implemented in different Eclipse plug-ins. It is remarkable that only the plug-in in charge of executing the model transformations is public and open source. All the remaining plug-ins that implement additional functionality (text editor, debugger, etc..) are closed source.

According to these licenses, the prototype which is presented in this thesis and is in charge of executing model transformations only makes use of the open source component which can be freely reused. This way, the core transformations engine of *medini QVT* has been packaged in our tool to provide a set of user-friendly interfaces that execute our model transformations as is described in detail in section 8.4.

3.6 SUMMARY

This chapter has briefly described the tools which are available in the market to carry out complex MDE processes. In this case, the tools which are interesting for the purposes of this thesis must be interoperable and must comply with the industrial standards for modeling, metamodeling and model transformations. In this sense, EMF and the tools that are built on top of it fulfill our expectations and will be used to implement and exploit the models and transformations proposed in this work. Regarding to the transformations engine, we have selected *mediniQVT* as it provides support for QVT–Relations which is a declarative and standardised language, and the source code can be reused under the terms of the EPL.

4

SOFTWARE PRODUCT LINES: DEALING WITH COMMONALITIES AND VARIABILITIES IN SOFTWARE FAMILIES

«Thinking is the hardest work there is, which is probably the reason why so few engage in it.»

— Henry Ford
Pioneer of the assembly-line production method, 1863–1947

The changing nature of technology leads us to need multiple versions of the same or similar application in short time periods. Due to cost and time constraints it is not possible for software developers to make a product from scratch for each new customer, so the reuse of software should be enforced. Because of that, Software Engineering must provide the tools and methods which allow us to develop a family of products with different capabilities and adaptable to changeable situations, in place of developing only a single product. Under these circumstances, the Software Product Line (SPL) concept arises with the aim of controlling and minimizing the high costs of the software development process. According to Clements and Northrop (2001),

> *"a SPL 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 that*

> *are developed from a common set of core assets in a prescribed way.*"

This approach is based on the creation of a design that can be shared among all the members of a family of programs within an application domain. This way, a design that has been done explicitly for a product can get benefit from the *core assets* (architecture, models, requirement specifications, components, code, test cases...) that can be reused in different products, reducing costs and development time.

SPLE aims to apply the principles of mass production introduced by Henry Ford in the automotive industry to obtain software applications. I. e., factories do not produce single products but families of similar products.

In short, we can say that a product line is a group of products that have a common set of features and vary only in some specific features. Features are an abstract concept to describe similarity and variability, and can be used to distinguish the products of a SPL. Each feature is an increase in product functionality. In section 4.2 we will describe in detail what features and feature modeling are.

From a practical standpoint, SPLs are one of the most successful approaches to software reuse, as they focus on developing families of systems which share a basic architecture. This way, SPLs provide an industrial approach to software development processes. Traditionally, SPLs aim to develop a framework to represent a family of products, which is adapted to develop individual products. A product family is a collection of similar products that share many features.

4.1 SOFTWARE PRODUCT LINE ENGINEERING

Software Product Line Engineering (SPLE), or *software product line practice* (Clements and Northrop 2001), is *the systematic reuse of core assets to assemble, instantiate or generate the multiple products that constitute a SPL*. The SPLE approach to software development involves changing the existing development process by introducing a distinction between the *domain engineering* and *application engineering* stage. Such a division is fundamental in the application of SPL techniques. We describe a domain as *a specialized body of*

knowledge, an area of expertise or a collection of related functionality.

In the *domain engineering* phase, all the information and knowledge about a particular domain is captured in order to create the reusable software assets. At this stage the family of products is analyzed to determine the commonalities (common requirements) and variabilities (product-specific requirements) among the members of the SPL, and next, a *reference architecture* of the SPL is designed. The *reference architecture* is the one that contains the components that are common to all the members of the family. This architecture also describes which optional components are required only by some members and how they can be configured. The formal description of this process (how to configure and assemble the different assets) is known as the *production plan*. Finally, all the software assets that will be used to produce software products are built and stored in the *baseline*. A *baseline* is a specialized database that stores software assets and facilitates their recovery and maintenance. Its aim is to ensure the availability of core assets to support the development of the SPL products.

SPLs are enacted in two stages: domain engineering *and* application engineering. *In the* domain engineering *stage the basic components (*core assets*) are developed, and a production plan is designed. In the* application engineering *stage the product is built using the* core assets *and the previously defined production plan.*

The *application engineering* phase is responsible for product development through the reuse of software assets using the designed *production plans*. The reference architecture is used as a reference model for building the products of the SPL. The *baseline* provides the required assets for the development of each new product.

Fig. 4.1 shows the three essential activities that should be carried out to develop a software product line:

CORE ASSET DEVELOPMENT represents the ongoing activities to develop reusable building blocks. Their inputs are the core assets used in the family of products and the production plan that indicates how to use these assets to assemble a final product.

PRODUCT DEVELOPMENT are the engineering activities to build products using reusable assets that were described in the production plan.

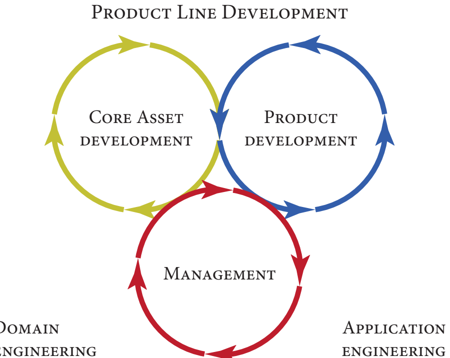


Figure 4.1: Essential Activities for Software Product Lines

MANAGEMENT represents activities of technical and organizational management.

These three activities are essential for the developments of a SPL and all of them are interrelated. Each one can be carried out in any order, leading to an iterative development. Table 4.1 summarizes the goals of these three activities both in the *domain engineering* phase and the *application engineering* phase as described by Clements and Northrop (2001).

4.2 DESCRIBING VARIABILITIES AND COMMONALITIES IN SOFTWARE FAMILIES

Software Product Lines aim to control and minimize the high costs of developing a family of software products. As described previously, this approach is based on the creation of a design that can be reused among all the members of a family of programs. But the key aspect that characterizes SPLs against other reuse techniques is that software reuse is planned since the development process is designed.

The first stage when developing a SPL is to perform an analysis in order to identify the commonalities and variabilities in the domain. To make things easy, it is desirable to have the results of the

	DOMAIN ENGINEERING
DOMAIN ANALYSIS	The variability of the domain is studied and analyzed. Usually, this study is carried out to identify the characteristics of the domain. A model with such elements is built (a feature model, see section 4.2).
CORE ASSET DEVELOPMENT	The core assets (reusable building blocks) are designed and implementd. This step not only captures the functionality of the domain, but also how the core assets can be extended should be defined.
PRODUCTION PLAN	This stage describes how the individual products should be assembled using the core assets.
	APPLICATION ENGINEERING
PRODUCT CHARACTERIZATION	The features that characterize the desired product should be selected.
PRODUCT SYNTHESIS	The *baseline* is queried and the needed core assets are retrieved to build the final product.
PRODUCT CONSTRUCTION	The selected core assets are processed following the production plan to obtain the final product. The production plan can specify that several tasks should be carried out (i. e., code generation, compilation, execution of programs, etc.).

Table 4.1: Development and application of a SPL

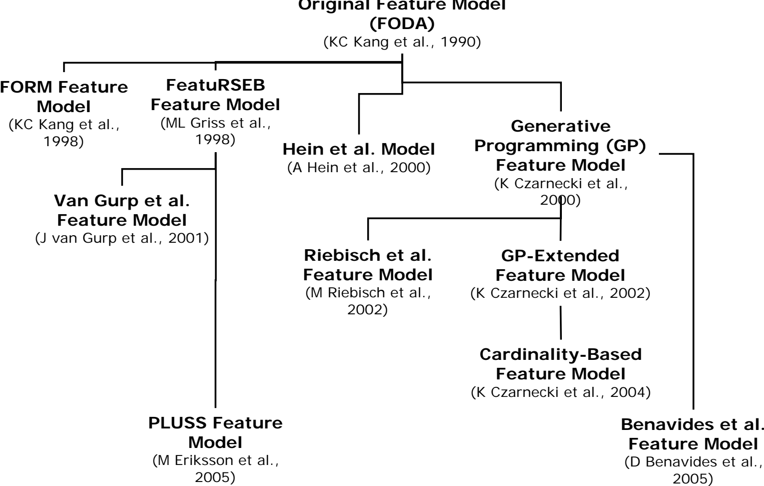


Figure 4.2: Feature model genealogy (Kang 2009)

analysis in an organized way, allowing the reuse of this analysis in the software development process. In 1990, a *method for discovering and representing commonalities among related software systems* was proposed by Kang et al. (1990). It is this context that the notion of *feature* arises.

Because of the current activity in the feature modeling and variability management community, several systematic reviews have been published recently (Chen et al. 2009; Kang 2009). From these reviews follows that the conferences which give more coverage to variability management and feature modeling are the *SPLC (International Software Product Line Conference)*, which already has 14 editions; and *ICSE (International Conference on Software Engineering)* which has 33 editions. For its part, the most relevant and active authors in this community are K. Czarnecki and D. Batory.

The most important works appeared in the field of SPLs have been studied, and throughout the following section we will summarize the main contributions made in the last 20 years in the feature modeling field. Starting with the proposal by Kang et al., next we will describe the main variations made to this initial proposal.

4.2.1 *Introduction*

Features can be described as *user-visible aspects or characteristics of the domain* according to Kang et al. (1990) or as *a distinguishing characteristic of a software item (e. g., performance, portability, or functionality)* as described by the IEEE 829-1998 standard (IEEE 1998).

The Feature-Oriented Domain Analysis (FODA) proposal presented by Kang et al. (1990) can be considered the most important contribution for feature analysis and management (Kang et al. 1998; Czarnecki and Eisenecker 2000; Kang et al. 2002), and subsequent contributions are strongly based and influenced by this work as later systematic reviews reflect (Chen et al. 2009). Fig. 4.2 presents the genealogy of the different contributions made in this field. In this figure Prof. Kang presents a summary about the evolution of feature models to celebrate the 20th anniversary of the advent of the feature modeling proposal.

*In feature modeling, systems are described in terms of features (*user-visible aspects or characteristics of the domain*). Feature models represent a tree-like structure of features which describe commonalities and variabilities among systems.*

Feature models are diagrams which express the commonalities and variabilities among the products of a SPL. These models organize the so-called features in a hierarchical structure. They describe a set of relationships among parent features and child features. The basic relationships between a feature and its children are: *mandatory* relationships (which represent the shared design), *optional* relationships, *OR* and different types of groups. Cross-tree relationships are also common to describe inclusion or exclusion constraints. Other extensions and variants have been proposed in different works. Next, we will describe the most important proposals which are related with our work.

4.2.2 *Classic feature models (FODA proposal)*

Classic feature models (FODA feature models) define only three kinds of relationships between a parent feature and its children:

MANDATORY Mandatory features represent features which are required to be included in a given product. These features represent the common aspects to all the members of the family. A line drawn between a child feature and a parent feature indicates that a child feature requires its parent feature to be present. In the original FODA notation mandatory features are those which are described with no special notation. For example, in Fig. 4.3a both features F and A are mandatory.

OPTIONAL Optional features are those that may be included or not in a given product of the family. In the original notation optional features are denoted by using a circle. In Fig. 4.3b feature F is mandatory (and must be included in any possible product) and feature A is optional, and may be included or not.

In FODA feature models the shared design is represented by the mandatory features. The variability can be described by using both optional features and alternative groups.

ALTERNATIVE This relationship can be defined among a parent feature and a set of children features. In this case, this relationship states that children features can be considered as a specialization of the parent feature, and only one child can be present in a given product of the family. Fig. 4.3c describes a feature model where only two possible products are possible, the first one is made up by the features F and A, and the second one is made up by the features F and B.

In addition to the previous relationships the so-called *composition rules* cab be defined between any pair of features:

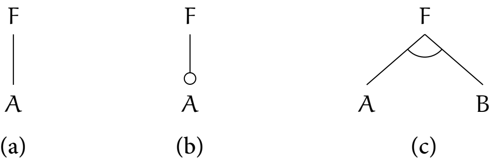


Figure 4.3: Example of FODA relationships

Listing 4.1: Textual representation for the FODA *excludes* and *requires* relationships

```
1 <feature1> ('requires' | 'mutex-with') <feature2>
```

MUTUAL DEPENDENCY (REQUIRES) This relationship states that one feature requires the existence of another one (i. e., they are interdependent).

MUTUAL EXCLUSION (MUTEX-WITH) implies that one feature is mutually exclusive with another one (they cannot coexist).

The original FODA proposal does not define a graphical representation for composition rules, rather, it provides a textual representation as shown in listing 4.1. Nevertheless, these relationships can be easily drawn in feature models by using an arc between the features involved. Fig. 4.5 shows an example of a typical representation. The exact notation can vary depending on the author by using dashed lines, different arrow ends, etc. As it can also be observed, in modern notations it is quite common to represent features as named boxes.

The semantics of all these relationships can be expressed by means of propositional formulas (Batory 2005); this way, it is possible to reason about the satisfiability of the feature model and its configurations.

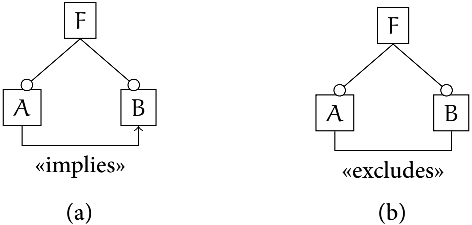


Figure 4.4: Example of the *implies* (4.4a) and *excludes* (4.4b) relationships

4.2.3 *FeatuRSEB and PLUSS feature models*

Griss et al. (1998) introduced *FeatuRSEB*, a new proposal to integrate feature models into processes of the Reuse-Driven Software Engineering Business (RSEB) approach (Jacobson et al. 1997). The RSEB is a software reuse technique where architecture and reusable subsystems are initially described by use cases, which are then transformed to object models. In the RSEB proposal, variability is captured by structuring use case and object models using explicit variation points and variants. However the RSEB does not provide explicit models which describe the commonalities for all the products of the family. This way, *FeatuRSEB* adds an explicit domain engineering phase and a explicit feature model to support domain engineering and component reuse.

FeatureRSEB *provides the same modeling primitives than* FODA*, plus a new kind of relationship to describe* XOR *groups.*

FeatuRSEB provides almost the same primitives to build feature models than the traditional FODA approach. The *FeatuRSEB* can describe *mandatory features*, *optional features*, *alternatives* (*XOR groups*) and *OR groups*. The last primitive is a new kind of group which allows to select one or more children features in comparison with the *alternative* relationship of the original FODA notation, which allows to select only one child. The *FeatuRSEB* notation also provides *requires* and *mutex-with* constraints.

The graphical notation proposed by *FeatuRSEB* is quite similar to the FODA notation. Mandatory and optional features are described respectively as in Figs. 4.3a and 4.3b

Figure 4.5: *XOR* (4.5a) and *OR* (4.5b) groups in *FeatuRSEB*

Groups of features are represented using rectilinear arcs, and *OR* and *XOR* groups are identified using black and white diamonds respectively. This way, Fig. 4.5a describes an alternative group equivalent to Fig. 4.3c, where only one child feature can be selected (A or B). In the case of Fig. 4.5b either feature A, feature B, or both A and B can be selected.

Eriksson et al. (2005) presented the *Product Line Use case modeling for Systems and Software engineering (PLUSS) toolkit*, which is based on *FeatureRSEB* proposal. However, this work uses feature models from a different point of view. Unlike *FeatuRESB*, where feature models play a key role in the development process, in the *PLUSS toolkit* they are only used as a *tool for visualizing variants in an abstract product family use case model.*

The semantics of the feature models that can be described in the *PLUSS toolkit* are exactly the same than in the *FeatuRSEB* approach. However, in the *PLUSS toolkit* the authors provide a different graphical notation, as shown in Fig. 4.6, due to limitations in their tool support. This way, features represented by a black and white circles are mandatory and optional respectively. Alternative features (*XOR*) are renamed as *Single Adaptor* features, and are represented by a gray circle with an "S" inside it. Finaly, *OR* groups are called *Multiple Adaptor* features, and describe the relation *at-least-one-out-of-many*.

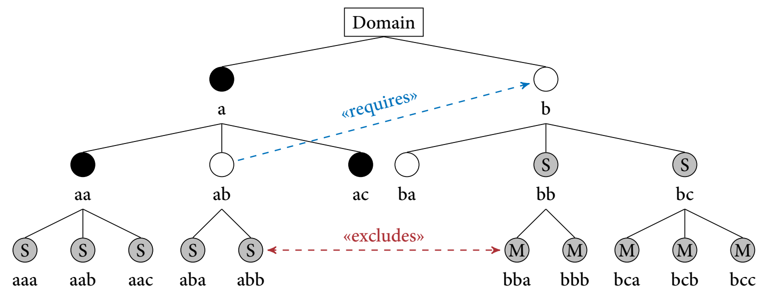


Figure 4.6: Example PLUSS feature model extracted from (Eriksson et al. 2005)

Typical *excludes* and *requires* relationships can also be described as shown in the figure.

4.2.4 *Cardinality-based feature models*

Cardinality-based feature models provide a more expressive notation than traditional FODA feature models. This thesis uses this kind of models to describe system's variability as next chapters show.

Cardinality-based feature modeling (Czarnecki et al. 2005a) integrates several extensions that have been contributed to the original

SYMBOL	SHORT DESCRIPTION
F	Mandatory feature (cardinality $[1..1]$)
F	Optional feature (cardinality $[0..1]$)
$[n..m]$ F	Feature with cardinality [n..m]. If $m > 1$ then F is a clonable feature
	Feature group with cardinality $\langle 1..1 \rangle$ (exclusive or group)
	Feature group with cardinality $\langle 1..k \rangle$, where k is the number of group elements (or group)
$\langle i..j \rangle$	Feature group with cardinality $\langle i..j \rangle$, where $0 \leqslant i \leqslant j \leqslant k$
F(T)	Feature with attribute type T

Table 4.2: Cardinality-based feature modeling basic primitives

FODA notation. Table 4.2 describes the main elements of the proposed modeling language. A cardinality based feature model is a hierarchy of features; the main difference with the original FODA proposal is that each feature has associated a *feature cardinality* which is expressed in a UML-like notation. Feature cardinality can be used as a general way to describe mandatory and optional features. Mandatory features have a lower bound equal to 1, and an upper bound equal to 1. In contrast, optional features have a lower bound equal to 0 and an upper bound equal to 1. Features can also have an upper bound higher than 1, which specifies how many clones (instances) of the feature are allowed in a specific product configuration. Cloning features is useful in order to define multiple copies of a part of the system that can be differently configured. Moreover, features can be organized in *feature groups*, which also have a *group cardinality*. Feature groups are a generalization of the notion of the alternative features (*XOR groups*) and *OR groups* proposed by previous approaches. Group cardinality restricts the minimun and the maximun number of group members that can be selected. Cardinality-based feature models also allow to specify an *attribute type* for a given feature. Thus, a primitive value (string, integer, etc.) for this feature can be defined during configuration, which is useful to define *parameterized features*.

Fig. 4.7 shows an example cardinality-based feature model. The model describes a configurable text editor. Notice that the *documentClass(String)* and the *ext(String)* features are clonable. Both features have also an attribute type (*String*, in both cases). The fea-

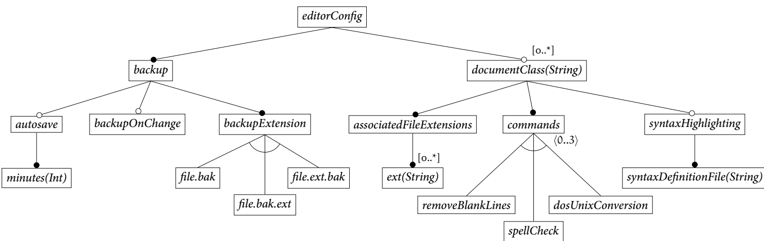


Figure 4.7: Text editor configuration example, extracted from (Czarnecki et al. 2005a)

ture *commands* has a group with cardinality ⟨0..3⟩ (which means that it is an *OR group*).

Cardinality-based feature models also allow to describe constraints between features such as the *implies* and the *excludes* relationships, which are the most common used ones (Czarnecki and Kim 2005). In classic feature models, the semantics of these constraints can be expressed by using propositional formulas (Batory 2005). However, this interpretation for feature models is not very adequate when dealing with cardinality-based feature models (Czarnecki and Kim 2005) since we can have multiple copies of the same feature. Therefore, it is necessary to clearly define the semantics of the constraint relationships in a context where features can have multiple copies, and features can have an attribute type and a value. In this case, we need more expressive approaches to (i) define constraints between features; and (ii) perform formal reasoning over the feature models and their constraints.

4.2.5 *Feature model configurations*

A configuration of a feature model can be considered as a valid set of instances of it.

A configuration of a feature model can be defined as *a valid set of instances of a feature model.* A configuration is made up from the mandatory features of the model, and a subset of *selected* optional features (for simple FODA models). In Fig. 4.8a an example feature model is represented. The model represents a system S, with two features A and B. The former is mandatory (i. e., feature A *must*

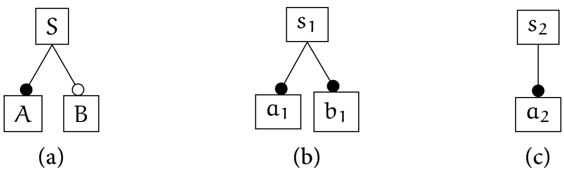


Figure 4.8: Example of a feature model (4.8a) and the two possible configurations that it represents (4.8b and 4.8c)

be included in every possible product of the product line), and the latter is optional (feature B *can* be included in a particular product or not). Thus, we have two possible configurations for this feature model, which are represented in figures 4.8b and 4.8c.

4.3 SUMMARY

SPL is a software reuse technique where a common design is shared among the different members of a family of products. In a SPL it is crucial to describe the variabilities and commonalities among such products in an explicit and understandable way. To achieve this, feature models have arisen as a suitable notation to describe such commonalities and variabilities. Feature modeling has been an important discussion topic in the SPL community, and a great amount of proposals have arisen in the last 20 years. Table 4.3 shows a summary of the most important graphical notations proposed for feature modeling. Notice that one of the most expressive notations is the *cardinality-based feature modeling* notation, which allows clonable features, feature groups with cardinalities and typed features (which can be used to express *parameterized features* and *feature attributes*).

"Pure" cardinality-based feature models use multiplicities to define optional and mandatory features. However, it is not unusual to find publications where the traditional notation is used as a visual shortcut (using black and white circles).

Like optional and mandatory features, OR and XOR groups can also be expressed using cardinalities in pure cardinality-based feature models.

	FODA	FEATURSEB	PLUSS	CARDINALITY-BASED
MANDATORY FEATURE	F F_1	F F_1	F F_1	F [1..1] F_1 or F F_1
OPTIONAL FEATURE	F F_1	F F_1	F F_1	F [0..1] F_1 or F F_1
CLONABLE FEATURE	—	—	—	F [n..m] F_1
XOR GROUP	F F_1 F_2 F_3	F F_1 F_2 F_3	F S S S F_1 F_2 F_3	F F_1 F_2 F_3
OR GROUP	—	F F_1 F_2 F_3	F M M M F_1 F_2 F_3	F F_1 F_2 F_3
GENERIC GROUPS	—	—	—	F ⟨i..j⟩ F_1 F_2 F_3
TYPED FEATURES	—	—	—	F(T)

Table 4.3: Summary of feature model notations

Part III

VARIABILITY VIEW ON MULTI-MODEL DRIVEN SOFTWARE PRODUCT LINES

SUMMARY

In this part we present our feature-based approach to manage variability in Multi-Model Driven Software Product Lines. Chapter 5 introduces what MMDSPLs are, and how they are related with the different views of a system. Chapter 6 describes the main issues that arise when feature models are used to describe the variability view in complex MDE processes. In chapter 7 we define our cardinality-based metamodel. We use MOF to describe feature models and we fully exploit them using the Eclipse Modeling Framework (EMF). Feature models can be enriched with complex model constraints that can be automatically checked by means of the pre-built OCL interpreters. All these EMF features allow developers to start a Software Product Line (SPL).

5

MULTI-MODEL DRIVEN PRODUCT LINE ENGINEERING

«Controlling complexity is the essence of computer programming»

— Brian Kernighan
Canadian computer scientist, developer of Unix and C, 1942–

Software Product Line Engineering (SPLE) enables the rapid development of product families. As discussed in chapter 4, a basic requirement is the management of the variability among family members. Therefore, an important factor in the design of a SPLs is feature modeling.

On the other hand, MDA is the ideal framework for the representation of software artifacts from the modeling point of view. Both SPLE and MDA provide benefits in software development. SPLE provides the methodology for the variability management both in the domain engineering and application engineering phases. For its part, MDA provides mechanisms for abstraction, modeling standards, persistence and data and metadata exchange. The proposal for the joint use of both approaches is what is known as MDPLE.

However, the power of the union between MDA and SPLE does not (only) come from the use and reuse of feature models to guide the process of gluing code snippets and other assets. The real power of

this approach is that feature models can be used and combined in complex MDE development processes. This way, all kinds of artifacts and models have their place in the design and implementation of the production plans in SPLs.

Thus, the proposal for variability management of this thesis is beyond the MDPLE. This work is in line with what we will call Multi-Model Driven Product Line Engineering (MMDPLE). The multi-modeling approach arises from the need of rigorously integrate different modeling languages in the software development process.

The use of multiple languages to describe different system views eases the understanding of the systems' structure and allow the specialization of developers.

The use of multiple languages can cover different domains and system views. In turn, it enables the specialization of developers and reduces the learning curve in the use of modeling and specification languages. Moreover, the separation of different concerns (i. e., a particular set of behaviors of a software system) in a multidimensional way alleviates the problems that can arise when designing software. One of the means to carry out a separation in dimensions is through the decomposition of a system in structures which represent the system views. As explained by Courtois (1985), *an understanding of the concept and properties of nearly decomposable structures is essential for an understanding of the behavior of systems with many scales and for solving the problems raised by their analysis*. The use of system views (which are described by using different models in a multi-model) pursues the goal of decomposing systems in order to make them understandable.

The main problem that comes up in the use of multi-models is how to maintain consistency between the different views (or submodels) and how to establish the relationships among them. In this sense, the declarative model transformations emerge as a key technology, as they allow the definition of equivalence relationships (using patterns) among the models that are part of a multi-model. Boronat et al. (2008) discusses in detail the formal foundations of multimodel languages as well as the role that model transformations play in this proposal.

In MMDPLE, a SPL will represent a software development process driven by different models, each one considering a different system

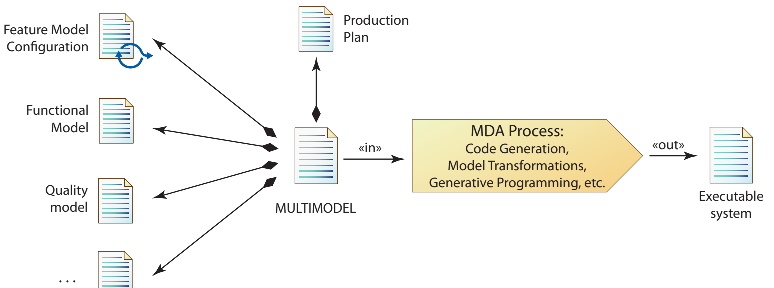


Figure 5.1: Multi-Modeling Driven Software Product Line Engineering

In a MMDSPL, and following the MDE principles, the core assets used in the development process are models and model transformations.

view. Fig. 5.1 shows a generic description of a multi-model driven SPL. There, several models describe the assets that participate in the software development process. Such models can describe variability aspects, functionality aspects, quality aspects, etc. Following the MDE approach, the production plan may also be defined as part of this multi-model. Specificalle, the production plan is the asset which describes how the different models are combined to provide the whole system view. This way, the production plan will mainly consist of declarative model transformations which will relate the different models of the multi-model.

5.1 SYSTEM VIEWS AND THE MULTI-MODEL

Several analogies have been drawn to describe what a system view is. Kruchten (1995) describes system views as the plans which describe a building from different perspectives, such as the floor plan or the elevation of a building; and Bass et al. (1998) illustrate views using the human body as an example. In this case, a view can be considered as the perception of the body that different medical specialist have when treating with a patient. I. e., doctors may be interested in the circulatory system (e. g., cardiologists) or the digestive system (e. g., gastroenterologists). Each specialist gets focused only in certain properties of the system (the human body) which are important for him/her. This way, a view is an abstraction as defined by Shaw (1984):

a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others.

How to deal with system's views has been a long standing issue in Software Engineering, and several works have proposed different models to represent them. Among the most relevant proposals we have Kruchten (1995), Hofmeister et al. (1999)—also know as the *Siemens* proposal—, and the Software Engineering Institute (SEI) proposal (Bass et al. 1998; Clements et al. 2003).

The SEI proposal considers three kinds of views as the basic structures to describe a software architecture, namely, the *modular style*, the *component–connector style* and the *allocation style*. The *modular style* (*modular view*) makes a partition of the system's functionality, where each part is considered as a *module*. The *component–connector style* (*component–connector view*) aims to model runtime aspects, such as concurrency and communication issues. Components and connectors are the elements used to represent the software entities which play a role while executing the system. Finally, the *allocation style* (*allocation view*) is used to relate the elements of the previous views with the elements existing in the environment where the system is executed.

The Limón Cordero's thesis proposes the use of the modular and the component–connector views metamodels to describe system views. This proposal is used as the starting point of our work.

In the context of the MDA and system's views we find the proposal of Limón Cordero (2010). This work, based on the SEI proposal, presents a framework to (i) describe different views of a system and; (ii) to establish the correspondences among them by using model management techniques. Limón Cordero proposes the use of two basic views: the *modular view* and the *component–connector view*. Limón Cordero provides the MOF-compliant metamodels to describe the system's views. Furthermore, the *allocation view* is provided by a set of QVT–Relations rules to describe the correspondences among the different views of a system.

As can be observed, this last proposal is very close to the definition of a multi-model. However, it does not consider the variability view, which is a fundamental view when dealing with software families in the context of a MDSPL. This thesis aims to cover this gap, and we

propose to add the system's variability view in order to be able to develop MMDSPLs.

5.2 VIEWS, MODELS AND METAMODELS IN MULTIPLE

This thesis presents MULTIPLE, a framework and a tool to describe and implement MMDSPLs. This way, different metamodels and DSLs must be provided to describe a system as a whole. The proposed framework aims to be extensible thanks to the capabilities provided by state of the art modeling tools. However, MULTIPLE provides *out of the box* support to define three main system views: the *variability view*, the *modular view* and the *component–connector view*. These views are models specified using different metamodels and domain-specific editors, which have been defined using MOF:

This thesis provides out of the box *support for four metamodels: variability, modular metamodel, component–connector and PRISMA.*

THE VARIABILITY METAMODEL allows to specify the variants in a software family. That is, it allows to describe which parts of the products are common to all the members and which parts may vary from a single product to another. This metamodel is inspired by the most relevant proposals in the feature modeling literature over the last 20 years. Chapters 6 and 7 describe in detail the variability metamodel, whose implementation is described in chapter 8, section 8.3.1.

THE MODULAR METAMODEL is based on the proposal made by Limón Cordero. It allows to describe a system in terms of *modules*, *functions* and *different types* of relationships that can be established between modules. This metamodel is specified in precise in Limón Cordero (2010, p. 144 sqq.). The implementation of this metamodel in MULTIPLE is presented in section 8.3.3.

THE COMPONENT–CONNECTOR METAMODEL is also based on Limón Cordero (2010, p. 150 sqq.). It allows to describe a system in terms of *components*, *connectors*, *services*, *roles*, *ports*

and *relations*. The implementation of this metamodel using MOF is presented in section 8.3.4.

THE PRISMA METAMODEL is a specialization of the *component–connector metamodel*. It allows to describe systems in terms of components and connectors, but adding support for aspects (Kiczales et al. 1997). We include an *Ecore* implementation of the PRISMA metamodel described by Pérez Benedí (2006). Thanks to the tools that are provided together with the specification of this metamodel it is possible to automatically generate code from PRISMA architectural descriptions—i. e., PRISMA-NET-MIDDLEWARE and PRISMA-MODEL-COMPILER (Pérez et al. 2008). Since, this metamodel has been precisely specified in previous literature (Pérez Benedí 2006; Pérez et al. 2008; Ali 2008; Costa-Soria 2011), only the implementation details of this metamodel are presented in section 8.3.5.

5.3 SUMMARY

This chapter has introduced what a multi-model is, and how using this concept Multi-Model Driven Software Product Lines (MMDSPLs) can be built. A MMDSPL is a SPL built using different modeling languages and notations, and based on the MDE principles. In this sense, the previous works done in the context of software systems' views provide a fundamental background to characterize the most relevant views that should be considered (Bass et al. 1998; Clements et al. 2003; Limón Cordero 2010). Specifically, this thesis is based on the proposal of Limón Cordero (2010), which provides a comparison among the different proposals made in the last years to characterize the software systems' views; moreover, it makes a proposal to describe and interrelate them by using the MDA. In MULTIPLE three views are considered: the *modular* and the *component–connector* views, which are based in previous literature; and the new *variability*

view, which previous works do not manage in a explicit way and is fundamental when dealing with SPLs.

All these views, together with the relationships among them, make up the multi-model which defines a SPL as shown in Fig. 5.1. In this thesis each view can be defined by using a particular DSL, and each view can be related with each other by using declarative model transformations. All these tasks are supported by the corresponding tool support as next chapters will demonstrate.

6

CARDINALITY-BASED FEATURE MODEL CONFIGURATION ISSUES

«The significant problems we have cannot be solved at the same level of thinking with which we created them.»

— Albert Einstein
German physicist and Nobel prize in Physics in 1921, 1879–1955

Feature modeling plays a key role in the definition of a SPL. As a modeling approach, it can be exploited by means of metamodeling standards such as MDA.

In this chapter we discuss the main issues that arise when trying to use feature models in a MDE process, and how to easily overcome them. Here, and in the remaining of this thesis, we present our approach to allow developers of SPLs to define, use and exploit feature models in a modeling and MOF-compliant metamodeling tool (such as EMF). Moreover, since the EMF framework provides several tools which permit us to enrich these models (by means of OCL expressions) and to deal easily with them (by using model transformations),we will use this framework to start a SPL.

6.1 INTRODUCTION

Feature models are diagrams which express the commonalities and variabilities among the members of a family of products. The cardinality-based feature modeling approach (see section 4.2.4) integrates several extensions that have been proposed to the original FODA notation.

A configuration of a feature model, can be defined as *a valid set of instances of a feature model* (see section 4.2.5). Expressed in terms of the Object-Oriented (OO) paradigm, the relationship between a feature model and a configuration is comparable to the relationship between a class and an object. Fig. 4.8 (page 58) shows an example feature model with its two possible configurations. Notice that the feature selection process (according to the defined constraints) is closely related with a copy mechanism, that is, a configuration of a feature model is a more restrictive copy of the original one that represents exactly one variant.

This definition is quite intuitive when dealing with "traditional" feature models (those that can be defined by using the original FODA notation). In this case, every instantiation of the elements of the feature model will follow the *singleton pattern* (Gamma et al. 1995), that is, every feature can have at most one instance. Fig. 4.8 (see page 58) show an example of this.

6.2 FEATURE MODELS, CONFIGURATIONS AND MOF

The MOF standard provides a suitable basis to define feature models and configurations.

The MOF standard, as presented in section 2.3, defines a strict classification of software artifacts in a four-layer architecture. Since it provides support for modeling and metamodeling, we can use MOF to define feature models by defining their metamodel. Fig. 6.1 shows the example feature model presented in Fig. 4.8 in the context of MOF. The EMOF language is represented in a simplified way in M3. In M2, the feature metamodel is represented by using the MOF language (also in a simplified way). The example feature model is shown in level M1 (left-most feature model).

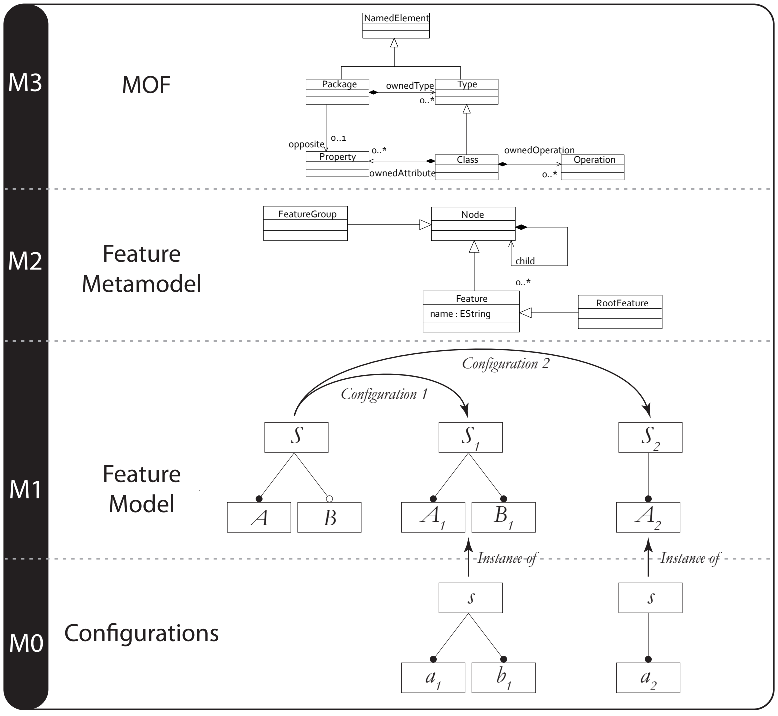


Figure 6.1: Configuration of FODA feature models in MOF

The EMOF *language is represented in a simplified way in the level* M3*. In the level* M2 *the metamodel for cardinality-based feature models is represented by using the* MOF *language (also in a simplified way). In the level* M1 *feature models are described. Some configurations are shown at level* M0*.*

A configuration of a feature model is built by selecting a subset of features of a model. Obviously, mandatory features must be always selected, and optional features may be selected or not. In practice, this implies that they can be removed from the feature model (*Configuration 2* in Fig. 4.8). This configuration mechanism (deletion of optional features) is done at the model level as instances are not usually identified as such in the feature modeling community. This can be done (and is somewhat intuitive) as features can have only a single instance and configurations at the M0 layer are *equivalent* to their corresponding feature models at the M1 layer.

When cardinality-based feature models were proposed, the same mechanism to build feature model configurations was adopted (Czarnecki et al. 2005b; Czarnecki et al. 2005a). In this case, when features have an upper bound greater than 1, they can be cloned. Thus, we can

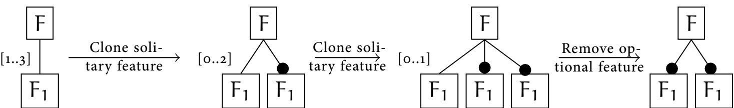


Figure 6.2: Staged configuration through specialization

have multiple *copies* of the same feature. This configuration steps are also done at model level until only one variant of the feature model is possible. Figure 6.2 shows an example of this.

Fig. 6.3 shows an example of the *configuration through specialization* mechanism in the context of MOF: several feature models are built until a feature model without variability (which identifies the configuration) is defined.

Configuration through specialization uses the refinement of feature models to describe model configurations. The configuration is described by a feature model which represents only one variant.

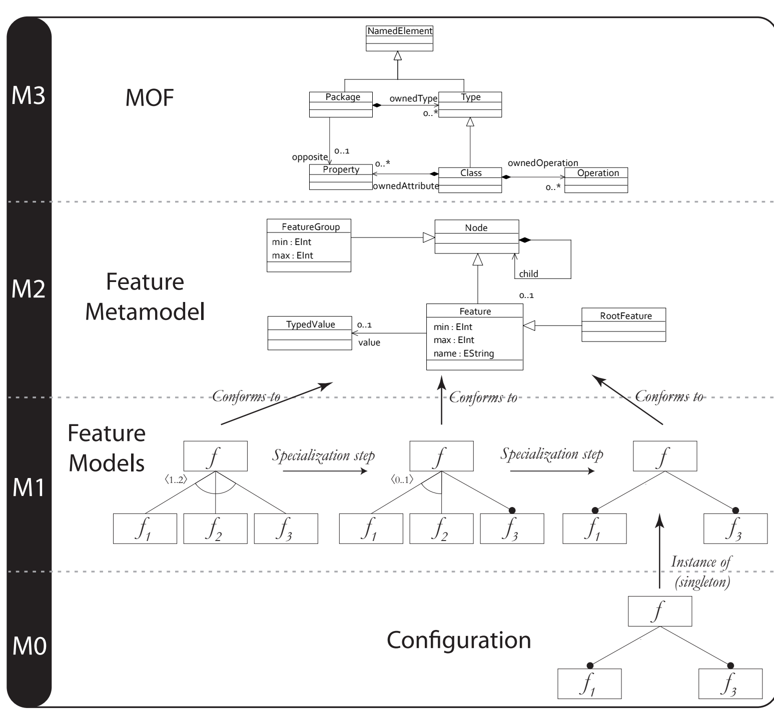


Figure 6.3: Configuration through specialization in the context of MOF

However, this approach to configure cardinality-based feature models presents some problems when using feature models in a MDE process. These problems come from the classical definition of configuration of a feature model (the set of features that are selected from a feature model). This definition mixes different levels of abstraction as it tends to define the configuration as a *copy* mechanism instead of as an *instantiation* mechanism. That is, the configuration is a refinement of the feature model instead of an actual instance of it.

The misconception in the instantiation process becomes more apparent when feature models also have *attribute types*. In this case, the relationship between a feature model and a configuration is more clear: it is similar to the relationship between a class with attributes and an object which defines its state in terms of the values for its attributes. Taking this into account, the configuration mechanisms are more similar and understandable as an *instance-of* relationship rather than as a *copy-and-refinement-of* relationship.

Mixing different levels of abstraction when dealing with software artifacts (such as the feature models) is also problematic when trying to apply existing MDE technologies. For example, to execute model transformations it is necessary to clearly define the involved artifacts and their relationships. A model transformation defines a set of rules among a set of metamodels. A set of candidate models (which conform to the metamodels) can be checked (or enforced) against the defined rules. Model transformations can be applied not only at metamodel/model level ($M2$ and $M1$ layers) but also to any $x + 1$ and x levels (such as the model/instance layers). Fig. 6.4 describes

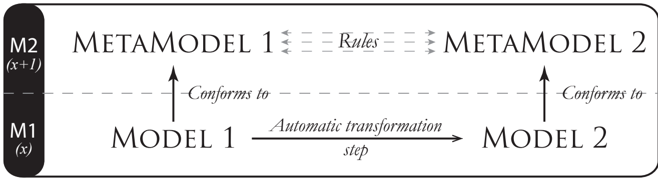


Figure 6.4: Example of a model transformation

In order to use feature models in MDE processes we need to clearly separate feature models and configurations in their respective levels.

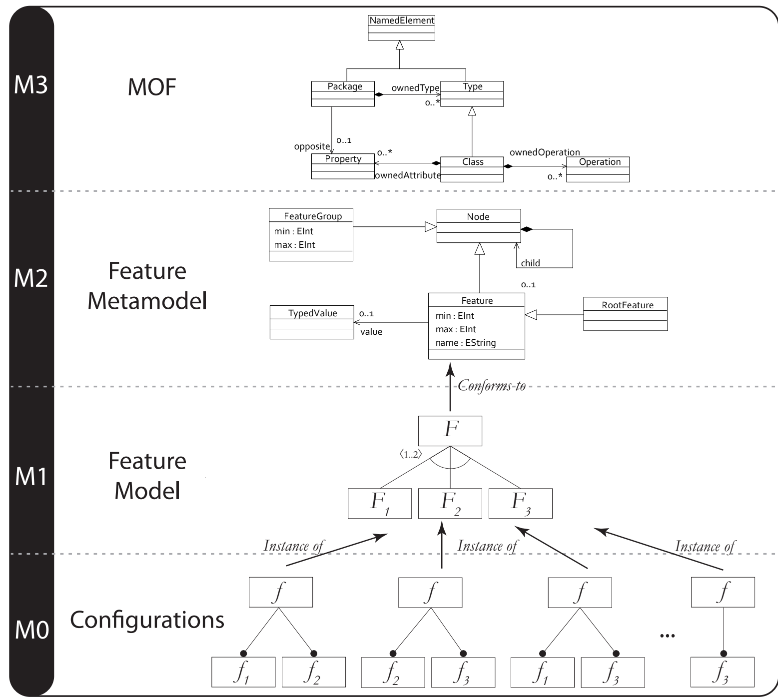


Figure 6.5: Definition and configuration of feature models in the context of MOF

an example transformation schematically with only two candidate models (and metamodels). The diagram describes the artifacts involved in a model transformation. In this example, a set of *Rules* defines how to transform from the metamodel *MetaModel 1* to the metamodel *MetaModel 2*. An initial model (*Model 1*, which conforms to *MetaModel 1*) is automatically transformed to obtain the *Model 2*, which conforms to *MetaModel 2*.

As a consequence of the above explanations, in order to integrate feature models in model transformations we must place each software artifact in its corresponding abstraction level. Fig. 6.5 shows where feature models and configurations should be placed in a well-defined MOF architecture. This way, a feature model which describes all the variants of the family of systems should be placed at the M1

layer; and any valid configuration described by the feature model should be placed at the M0 layer.

Moreover, when features can be cloned and such features can have an attribute type a new issue comes up: we can not represent feature models as propositional formulas as in FODA. Propositional formulas only allow two possible values: `true` or `false`, i. e., the feature is selected or not. In cardinality-based feature models we need to express how many clones of a feature exist, which are the values of the attributes (which can be strings, integers, ...), etc. In this case, we need to use more expressive languages that allows us (i) to deal with sets of features (i. e. n copies of feature F); and (ii) to deal with typed variables which values can be unbound in order to easily represent attribute types.

6.3 DESCRIBING FEATURE MODEL CONFIGURATIONS AS INSTANCES

The previous section has shown the importance to describe a configuration as an instance of a feature model. MOF provides the conceptual background to describe feature models and configurations. Now, we must find the way to implement such approach in a modeling environment. EMF, the industrial framework that can be considered as an implementation of the MOF standard, can be used to achieve this goal.

Ecore, the EMF metamodeling language, can be placed at layer M3 of the four-layer MOF architecture. Using *Ecore*, developers can define their own models which will be placed at the metamodel layer (M2). An example of such metamodels is the one to build cardinality-based feature models. Finally, these *Ecore* models can be used to automatically generate graphical editors which are capable of building *instance models*, which will be placed at M1 layer. In the case of feature modeling, these *instance models* are the feature models. Fig. 6.6 shows this architecture.

A drawback of most of the modeling frameworks which are available today is that M0 is empty. Specifically, EMF provides a modeling

The Ecore *language is placed at the* M3 *layer, the* Ecore *models (such as the metamodel for cardinality-based features models) are placed at the* M2 *layer, and model instances (feature models) are placed at the* M1 *layer. It is noteworthy that* EMF *can not represent more than 3 layers.*

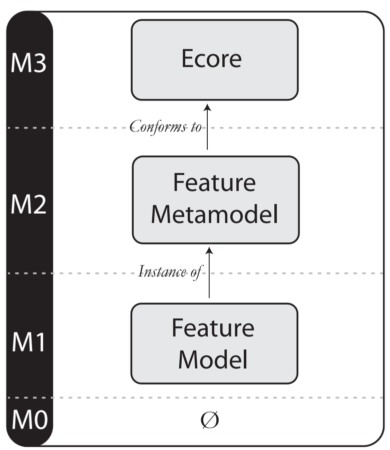


Figure 6.6: EMF and the four-layer architecture of MOF

language to define new models and their instances, but it only covers two layers of the MOF architecture: the metamodel (M2) and the model (M1) layers. However, in the case of feature modeling we need to work with three layers of the MOF architecture: metamodel (cardinality-based feature metamodel), model (cardinality-based feature models), and instances (configurations).

Fig. 6.7 shows how we overcome this drawback to provide complete feature modeling support in MULTIPLE: we define a model-to-model transformation to convert a feature model (i. e. the model represented by *Feature model* which can not be instantiated) to an *Ecore* model—which we call the *Domain Variability Model (DVM)*—, that represents the *Feature model* as a new class diagram. Thus, it is possible to represent a feature model at the metamodeling layer, making the definition of its instances possible. This allows us to take advantage of EMF again, and automatically generate the editors to define feature model configurations in MULTIPLE, and validate them against their corresponding feature models thanks to their new representation, the DVM. Moreover, as the DVM is an *Ecore* model (a simplified UML class diagram) we automatically obtain support to check complex constraints (by using OCL) over the feature mo-

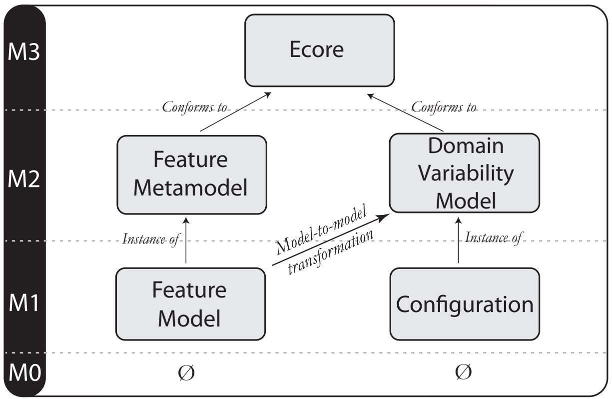


Figure 6.7: Dealing with feature model configurations in EMF

DVMs are a key artifact to allow the definition of feature model configurations in current modeling and metamodeling tools. A DVM is a class diagram which can be considered equivalent *to a feature model.*

del configurations. Chapter 7 describes how this is implemented in MULTIPLE in detail.

6.4 SUMMARY AND CONCLUSIONS

Feature models are used to define the variability of a family of products. A specific variant of a feature model is described by a *feature model configuration*. Using modeling and metamodeling terminology, a feature model configuration is in turn an instance of a feature model. However, traditionally the feature modeling community has considered a feature model configuration as a refined feature model instead of as an instance (the so-called *configuration through specialization* mechanism). This chapter has summarized the main issues that this idea implies. These problems become more apparent when dealing with cardinality-based feature models.

Configuration through specialization implies that a configuration is represented by a feature model which describes only one variant. This keeps us from using feature models in MDE processes, as we need to place software artifacts in the right levels of MOF. Representing configurations as actual instances of feature models allows us to describe in a more natural way the relationship between them. Furthermore, using the proper representation for feature models

and their configuration eases the validation tasks: the *instance-of* relationship guarantees the coherence between the elements at the M0 and the M1 layers.

Cardinality-based feature models also have some characteristics that make them quite different to classic feature models (FODA). Specifically, traditionally feature models were analysed using propositional formulas. However, cardinality-based feature models can have attribute types, which can not be described by this kind of logic. Thus, we need other kind of languages to analyse feature models and describe model constraints.

To overcome all this issues in MULTIPLE we propose to use an industrial modeling framework to build feature models and their configurations. However, some technical issues arise. Nevertheless, these issues can be easily solved by means of model transformations. This way, our proposal consists of transforming a feature model to a Domain Variability Model (DVM). Such diagrams allow us to easily define configurations as real instances, we can integrate them in complex MDE processes which use model transformations, and we can use existing technologies and constraint languages to analyse feature models and their configurations.

7 USING FEATURE MODELS IN MODEL-DRIVEN ENGINEERING PROCESSES

> *«Fundamental progress has to do with the reinterpretation of basic ideas»*
>
> — Alfred North Whitehead
> English mathematician and philosopher, 1861–1947

The basis of variability management in MULTIPLE is the cardinality-based feature metamodel, which permits to define feature models. As a result of state of the art review presented in section 4.2, we have decided to design a metamodel to define (a variant of) the cardinality-based feature models. The proposed metamodel can be considered a superset of the most relevant proposals for feature modeling. This chapter describes in detail our metamodel proposal, and how this metamodel can be used to define feature models and feature model configurations.

7.1 PROCESS OVERVIEW

As explained in the previous chapter, the use of feature models in MDE processes using nowadays modeling tools is not straightforward. The main problem that arises is the inability of these tools to deal

with software artifacts in more than two layers at the same time (i. e., they only support the *instance-of* relationship). This thesis proposes to use model transformations to overcome this drawback.

The use of model transformations has an impact in the process of defining a feature model and a feature model configuration, and such impact should be considered. Fig. 7.1 describes such process using Software & Systems Process Engineering Meta-Model (SPEM) (OMG 2008c). First, the *domain engineer* defines a feature model (which conforms to the feature metamodel described in section 7.2). Such feature model is used—together with the *Feature Model (FM) to DVM QVT transformation*—as an input of the *Obtain Domain Variability Model* task. such task, which is automatically executed, generates a DVM. Using the DVM the *application engineer* performs the *Feature Selection* task. In such task is when the *application engineer* selects the desired product of the SPL, and as a result, obtains a *Feature Model Configuration*.

Next sections detail each one of these tasks: section 7.2 describes the feature metamodel, which in turn determines how a feature model is defined; section 7.3 explains how a DVM is obtained from a given feature model; and finally section 7.4 describes how feature model configurations are defined by using the DVM.

The use of feature models has an impact in a software development process. The figure shows how a feature model can be used in a MDE process.

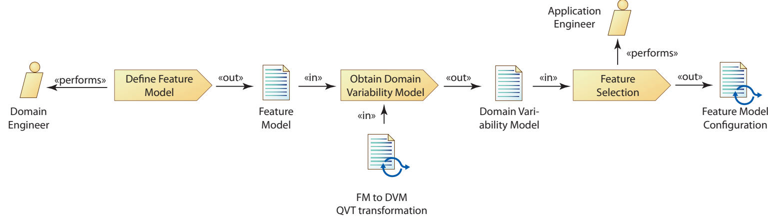


Figure 7.1: Feature models and MDE: process overview

7.2 CARDINALITY-BASED FEATURE METAMODEL

In our proposal we have decided to represent explicitly the relationships between features. Thus, our metamodel represents in an uniform way the hierarchical relationships and the restrictions between features. Table 7.1 classifies and summarizes the types of relationships that the feature metamodel is able to represent.

7.2.1 *Feature models structure*

Relationships represented in table 7.1 classify relationships in two orthogonal groups:

- *Vertical vs. horizontal relationships.* Vertical relationships define the hierarchical structure of a feature model and horizon-

	Vertical (hierarchical) relationships		Horizontal relationships
Binary relationships	Mandatory	[1..n]	Biconditional
			Implication
	Optional	[0..n]	Exclusion
			Use [0..n]
Grouped relationships	Generic	$0 \leq j \leq k \leq m$ [j..k]	
	XOR	$0 \leq j \leq 1$ [j..1]	
	OR	$0 \leq j \leq 1 < k \leq m$ [j..k]	

*where m is the number of childs

Table 7.1: Cardinality-based feature metamodel: proposed types of relationships between features

In contrast to other proposals, we propose to represent the different relationships between features in an explicit way. Relationships can be binary or grouped; and vertical or horizontal.

tal relationships define dependecies and restrictions between features.

- *Binary vs grouped relationships.* Binary relationships define relationships between two single features. In turn, grouped relationships are a set of relationships between a single feature and a group of childs.

Given this classification, the following relationships exist:

- *Binary and vertical relationships.* This relationships define structural relationships between two single features. In our approach, they represent a *has_a* relationship between a parent and a child feature. They can be mandatory and optional depending on the lower bound value. The upper bound (n) can be on both cases 1 or greater than 1, and indicates how many instances of the child feature will be allowed.

- *Grouped and vertical relationships.* Grouped and vertical relationships are a set of binary relationships where the child features share a *is_a* connotation with respect to their parent feature. A group can have an upper and a lower bound. These bounds specify the minimun and the maximun number of features that can be instantiated (regardless of the total number of instances).

- *Binary and horizontal relationships.* These relationships are specified between two features and do not express any hierarchical information. They are explained in the following section (sect. 7.2.2).

7.2.2 *Feature model constraints*

As was pointed out in section 6.1, it is quite common in feature modeling to have the possibility to define model constraints in order to describe more precisely which configurations should be considered

as valid. Typically, these constraints are described by means of implication or exclusion relationships. This kind of relationships are the *binary and horizontal relationships* that our metamodel provides.

The *binary and horizontal relationships* are specified between two features and they can express constraints (coimplications, implications and exclusion) or dependencies (use). The first group applies to the whole set of instances of the involved features, however, the second one allows us to define dependencies at instance level, i. e.:

The classical constraints that can be defined in feature models (requires, excludes) are classified as binary and horizontal relationships *in our proposal.*

- *Implication (*$A \longrightarrow B$*)*: If an instance of feature A exists, at least an instance of feature B must exist too.
- *Coimplication (*$A \longleftrightarrow B$*)*: If an instance of feature A exists, at least an instance of feature B must exist too and vice versa.
- *Exclusion (*$A \times\!\!\longrightarrow\!\!\times B$*)*: If an instance of feature A exists, can not exist any instance of feature B and vice versa.
- *Use (*$A \dashrightarrow B$*)*: This relationship will be defined at configuration level, and it will specify that an specific instance of feature A will be related to one (or more) specific instances of feature B as defined by its upper bound (n).

The classical requires *relationship corresponds to our implication relationship. The* excludes *relationship corresponds to our exclusion relationship.*

Besides these kind of relationships that describe coarse-grained restrictions, our metamodel provides capabilities to describe fine-grained restrictions. To describe these fine-grained restrictions we propose a constraint language, called *Feature Modeling Constraint Language (FMCL)*. FMCL is a formal language without side-effects (does not modify the model instances) whose syntax is based on the widely known OCL and its semantics are defined by a set of patterns that describe the equivalences between FMCL expressions and OCL expressions.

7.2.3 *Cardinality-based feature metamodel in MOF*

Fig. 7.2 shows our feature metamodel. Such metamodel has been defined taking into account that every element will have a different

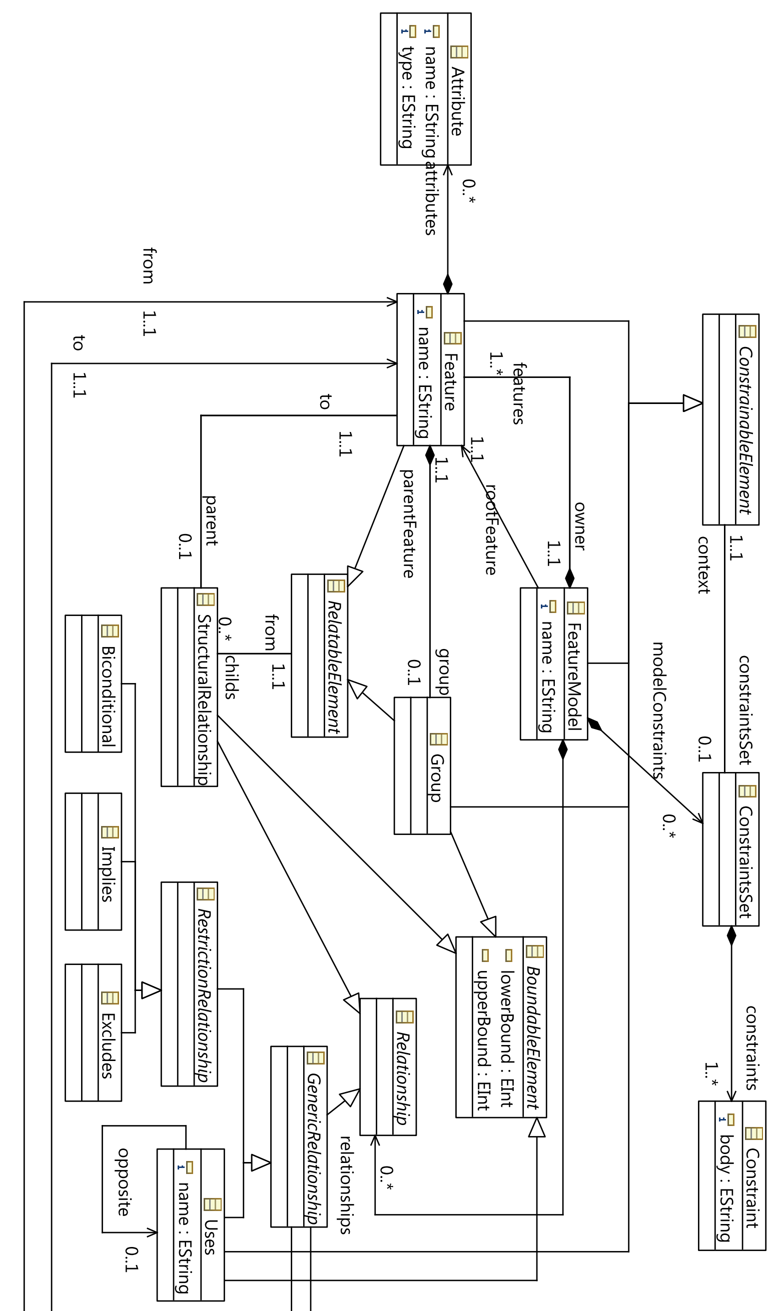


Figure 7.2: Cardinality-based features metamodel

graphical representation. This way, it is possible to automatically generate the graphical editor to draw feature models based on such metamodel. In that figure, a feature model is represented by means of the FeatureModel class, and a feature model can be seen as a set of Features and the set of Relationships among them. A feature model must also have a root feature, which is denoted by means of the rootFeature role.

Binary relationships in table 7.1 are represented in the features metamodel as descendants of the Relationship class. Class StructuralRelationship represents the so called *Vertical* relationships and GenericRelationship represents the *Horizontal* ones. StructuralRelationships relate one parent RelatableElement (a Feature or a Group) with one child Feature. A Group specifies that a set of StructuralRelationships should be considered as a group.

We have specified our metamodel for feature modeling using MOF. In our proposal, a feature model can be considered as a set of features and the set of relationships among them.

Complex model constraints expressed in FMCL are stored in a Constraints Set instance, and can be applied to any subclass of the abstract class ConstrainableElement (context role), i. e., FeatureModel, Feature, Group or Uses. The restrictions are expressed as a textual expression (body attribute of the Constraint class).

It is noteworthy to point out two slight differences of our approach with respect to the classical cardinality-based feature models. First, we represent feature multiplicities at relationship level instead of at feature level (by means of the BoundableElement class). This allows us to easily define mandatory and optional relationships explicitly. Second, features can not have an attribute type. In turn, this information is expressed in terms of feature attributes. Feature attributes express information which is complementary to a feature and can be used to describe *parametric features*.

7.2.3.1 *Example feature model*

Fig. 7.3 shows an example feature model using the proposed notation. The feature model describes a simple product line for cars. A car must have four wheels (of a given radius), one engine (of an specific power in watts) and a transmission (which can be manual or automatic). As

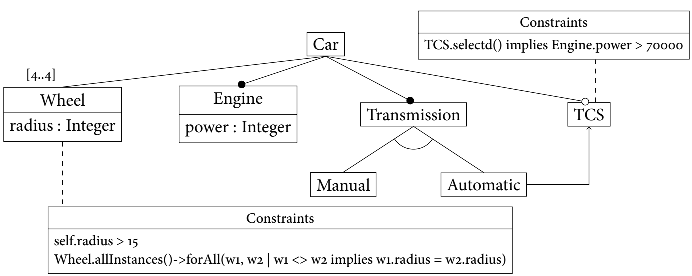


Figure 7.3: Example cardinality-based feature model

an optional equipment the car can have a Traction Control System (TCS). The feature model also describes four constraints: the arrow between the feature TCS and Automatic states that if an automatic transmission is selected, the TCS must be selected too; the annotation attached to the TCS feature states that the TCS can only be selected if the power of the engine is higher than 70.000 watts; and finally, the annotation attached to the Wheel feature specifies that the radius of the instances of the wheel must be higher than 15 inches and that all the wheels must be of the same size.

7.3 THE DOMAIN VARIABILITY MODEL

The DVM is the main artifact which allows using feature models in complex MDE processes. The equivalence between a feature model and a DVM can be established by a set of correspondence patterns.

The DVM is a class diagram (an *Ecore* model) whose instances are equivalent to the configurations of a feature model. It is intended to ease the definition of feature models configurations in EMF as was explained in section 6.3. This model can be automatically generated by means of a model-to-model transformation. The following paragraphs describe the transformation which converts a feature model, to a DVM (expressed as an *Ecore* model that can be instantiated).

As feature models describe not only the structure of the features but also the relationships and restrictions among them, it is necessary to define rules to generate both the structure of the DVM and the restrictions that apply to it. First, the structure of the DVM is defined by means of *Ecore* containment references and inheritance relationships; and second, restrictions are defined by means of OCL

expressions. These OCL expressions are included on the DVM itself by means of EAnnotations. These EAnnotations are automatically used in next steps by our prototype to check that configurations are valid.

7.3.1 *The structure of the DVM*

The transformation regarding to the structure of the DVM is almost a one-to-one mapping. For each Feature of the source model an *EClass* (with the same name) is created. All the classes are created inside the same EPackage, whose name and identifier derives from the feature model name. Moreover, for each feature Attribute, an EAttribute in its corresponding EClass is created in the target model. Any needed EDataType is also created.

The mappings between a feature model and a DVM are defined by a set of QVT–Relations declarative rules.

Regarding to the relationships, for each StructuralRelationship from a parent Feature, a *containment* EReference will be created from the corresponding EClass and for each Group contained in a Feature a *containment* EReference will be created from the corresponding EClass. This EReference will point to a new abstract class, whose name will be composed by the Feature name and the suffix "Type". Additionally, an EClass will be generated for each Feature belonging to a Group. Moreover, each one of these EClasses inherit from the abstract *EClass* that has been previously created.

Following the MDA guidelines, the mappings described previously informally have been clearly defined by using the Relations language defined in the QVT standard (OMG 2008a). Next, the relations are described using the graphical notation for QVT–Relations. Appendix A contains the full textual specification of the QVT transformation.

7.3.1.1 Feature2Class *relation*

The *Feature2Class* relation (Fig. 7.4) describes the mapping between Features and Classes. This relation is top-level, i. e., it is enforced on its own (it is not a pre- or post-condition for another relation).

The rule specifies that, given a feature model (whose name is modelName) and a feature (whose name is featureName), an EPackage containing an EClass must exist in the target domain. The EPackage name, nsPrefix and nsURI must match the feature model name. The EClass name must match the feature name.

7.3.1.2 FeatureAttribute2ClassAttribute *relation*

The *FeatureAttribute2ClassAttribute* relation (Fig. 7.5) is a top-level rule which specifies the mappings between feature attributes and class attributes. Thus, for each FeatureModel, an EPackage with the same name (and identifiers) must exist. For each feature Attribute an EAttribute with the same name must also exist. Such attributes are contained in their corresponding elements, i. e., feature attributes are contained in a Feature and EAttributes are contained in an EClass. These elements (Features and EClasses) must have the same name.

Finally, every feature Attribute must have a type. For each feature attribute type, an EDataType must exist in the declared EPackage. Such EDataType must correspond to a *Java* type contained in the `java.lang` package.

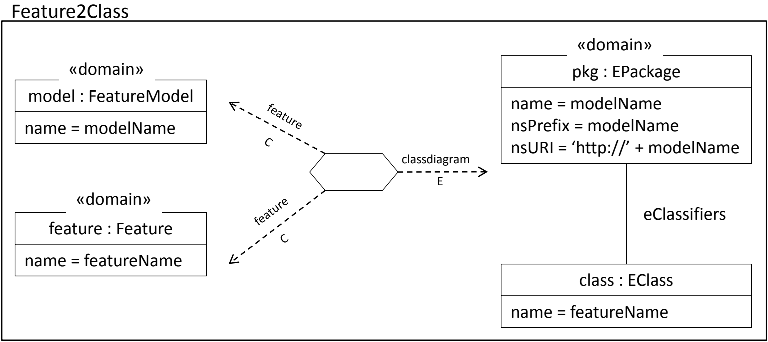


Figure 7.4: *Feature2Class* relation

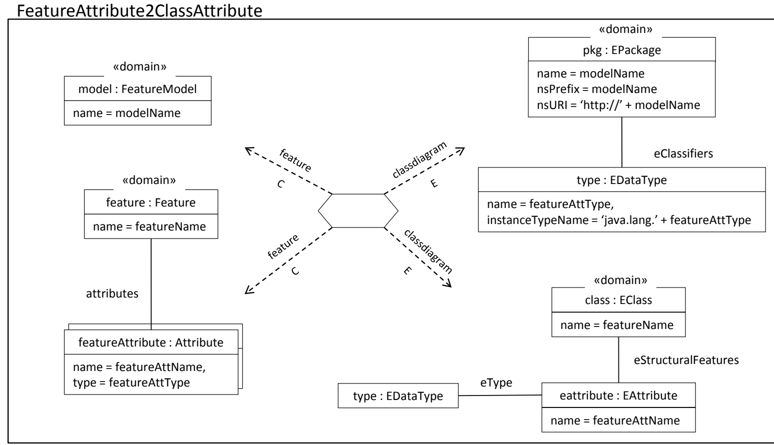


Figure 7.5: *FeatureAttribute2ClassAttribute* relation

7.3.1.3 StructuralRealtionship2Reference *relation*

This top-level rule (see Fig. 7.6) transforms the binary relationships between two Features to a *containment* EReference between two EClasses. As the previous rules, it checks that a Feature Model exists, and enforces that an EPackage with the same name also exists.

The rule also checks that for each StructuralRelationships between two Features whose names are featureName and childFeatureName, an EReference with the same name between two EClasses must exist. The names of the EClasses must be featureName and childFeatureName. The lower and upper bound of the EReference must be the ones specified by the StructuralRelationship (lowerBound and upperBound). Finally, the containmet attribute of the EReference must be `true`.

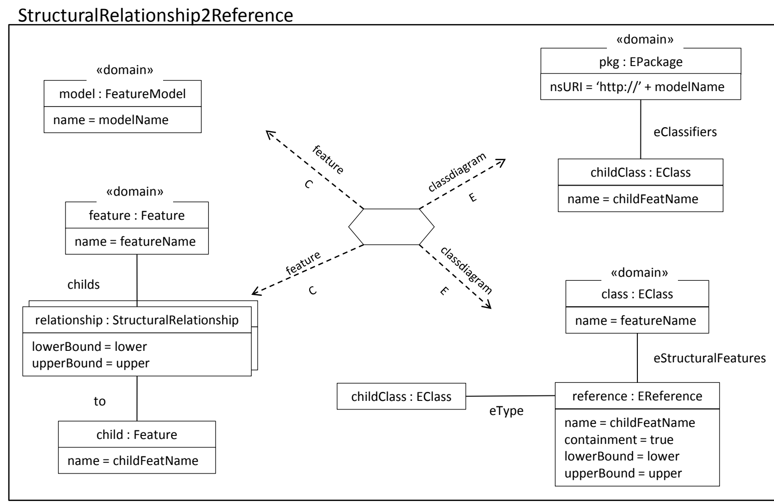


Figure 7.6: *StructuralRealtionship2Reference* relation

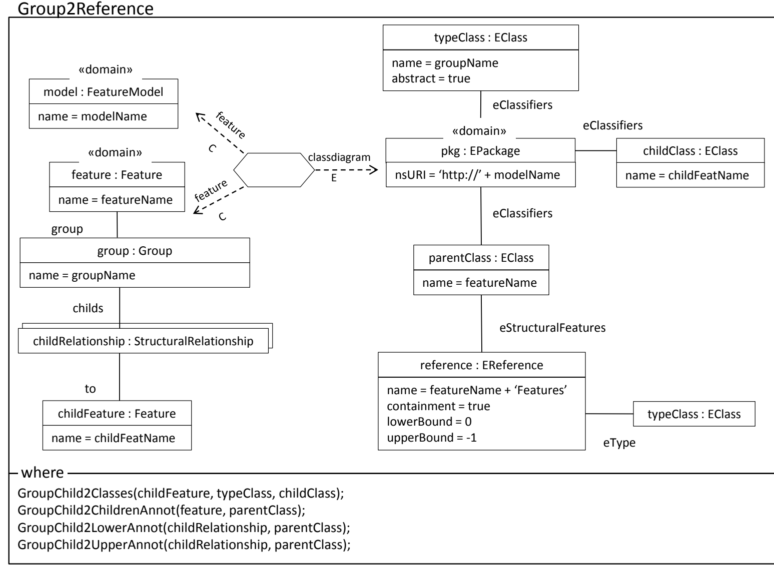


Figure 7.7: *Group2Reference* relation

7.3.1.4 Group2Reference *relation*

The *Group2Reference* top-level relation is in charge of transforming a Group of Features to a set of EClasses, EReferences and EAnnotations. This rule has several post-conditions, *GroupChild2Classes*, *GroupChild2ChildrenAnnot*, *GroupChild2LowerAnnot* and *GroupChild2UpperAnnot*, which are explained next.

Fig. 7.7 shows the object template which is applied to transform a Feature Group to the class diagram domain. First, as usual, it checks that both a Feature Model and an EPackage are found in the source and target domains with the same identifiers. It also checks that for each Feature which contains a Group, two EClasses must be present in the target domain. The first EClass must have the name of the feature (featureName). The second class must be an abstract class whose name must match the Group name (groupName). As was explained in page 86, grouped and vertical relationships denote an *is_a* relationship between a parent feature and its children. This way, this EClass is created to make explicit the *is_a* relationship mentioned before, as every child feature will inherit from it. The inheritance relationship is enforced when the *GroupChild2Classes* relation is applied. The abstract EClass is referenced from the parent EClass by means of a *containment* EReference. The lower and upper bound of the EReference are set to 0 and -1 respectively, which are the generic values to describe a *zero to many* elements multiplicity. The actual values of the different relationships are checked using OCL constraints. Such constraints are created by the post-conditions of the rule.

Finally, a Group in the source domain points to a set of child Features. For each one of these child features, an EClass with the same name (childFeatName) must exist in the target domain.

7.3.1.5 GroupChild2Classes *relation*

The *GroupChild2Classes* (Fig. 7.8) is executed as a post-condition of the *Group2Reference* and creates an inheritance relationships be-

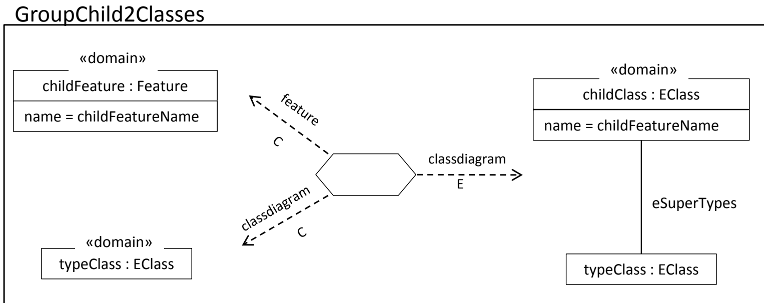


Figure 7.8: *GroupChild2Classes* relation

tween two EClases: childEClass and typeClass. The name of the child *EClass* matchs the name of an already created Feature.

7.3.1.6 GroupChild2ChildrenAnnot *relation*

One of the main differences of our proposal with respect to the traditional cardinality based feature models is how the group cardinalities are specified. I. e., we differentiate between the group cardinalities and the child cardinalities. When a group cardinality is specified, it restricts how many features can be instantiated (regardless of the number of instances of the feature), and the number of instances is restricted by the child cardinality.

Fig. 7.9 shows and example feature model. In it, an exclusive or group is defined (i. e., the group cardinality is [1..1]). This means that only one child feature can be instantiated (B or C, but not both). However, features B and C are clonable, and, as such, several in-

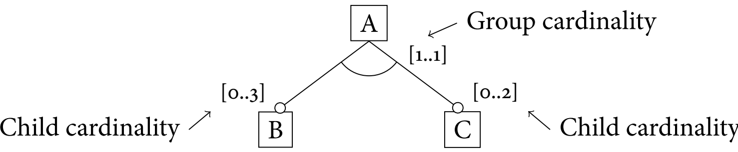


Figure 7.9: Cardinalities in feature groups

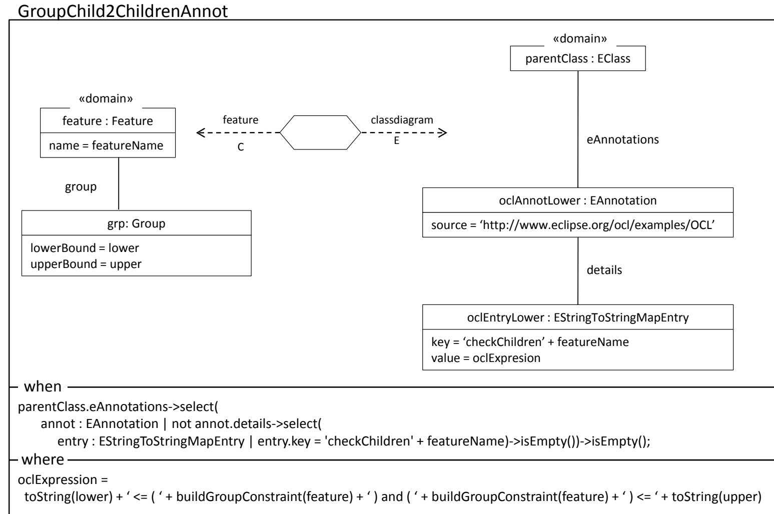


Figure 7.10: *GroupChild2ChildrenAnnot* relation

stances can be created (up to 3 instances of B and up to 2 instances of C).

Fig. 7.10 shows the *GroupChild2ChildrenAnnot* relation. It checks (and enforces) that for each Feature containing a Group a corresponding EAnnotation must exist in the parent EClass. To identify the EAnnotation the featureName will be used. The content of the EAnnotation will be an OCL expression. The `when` clause specifies the pre-condition of the relation, and states that the rule will be only applied if the EClass does not contain an EAnnotation with the same key attribute. The `where` clause builds the OCL expression which specifies that the number of EClasses with instances must be between the lowerBound and upperBound values of the Group. To count the number of EClases with instances the `buildGroupConstraint(...)` query is used (see Listing 7.1).

Finally, the Listing 7.2 shows the `toString(...)` function, which is able to convert an integer to string using OCL. This function is necessary because OCL is a strong typed language, and does not provide any built-in function to perform typecasting.

Listing 7.1: *buildGroupConstraint(...)* OCL query

```
1  query buildGroupConstraint(parentFeature : Feature) : String
       {
2    parentFeature.group.childs->iterate(
3      -- We iterate for each relationship contained in the
         group
4      relationship : StructuralRelationship;
5      -- The text of the OCL expression is stored in the "
         result" var
6      result : String = ''
7      | -- Starting from here, the body of the loop
8      result.concat(
9          '(if self.' + parentFeature.name + 'Features->select(
               f : ' + parentFeature.group.name + ' | f.
               oclIsKindOf(' + relationship.to.name + '))->
               notEmpty() then 1 else 0 endif) + ')
10     ).concat('0')
11 }
```

Listing 7.2: *toString(...)* OCL query

```
1  query toString(number : Integer) : String {
2    -- We define the following expression to translate an
         Integer to
3    -- String. In this way, we avoid to include any external
         library/method
4    -- to perform the conversion.
5    if number >= 0 then
6      OrderedSet{1000000, 10000, 1000, 100, 10, 1}->iterate(
7      -- We will support numbers <= 999.999
8      -- If greater numbers are needed, more powers of ten can
         be added
9        denominator : Integer;
10       s : String = ''|
11       let numberAsString : String = OrderedSet{'0','1','2','3
           ','4','5','6','7','8','9'}
12       ->at(((number div denominator) mod 10) + 1)
13       in
14         if s='' and numberAsString = '0' then
15           s
16         else
17           s.concat(numberAsString)
18         endif
19         )
20   else
21     '-'.concat(toString(-number))
22   endif
23 }
```

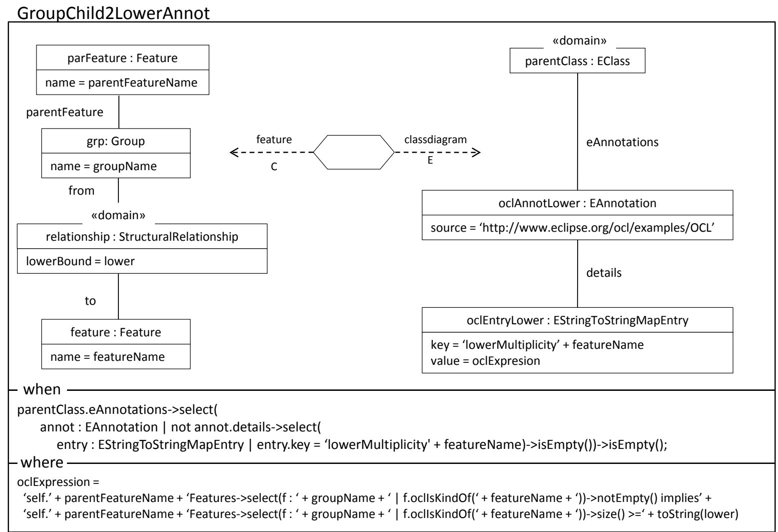


Figure 7.11: *GroupChild2LowerAnnot* relation

7.3.1.7 GroupChild2LowerAnnot *relation*

As explained before, in our proposal we diferentiate between group cardinalities and child cardinalities. As the *GroupChild2ChildrenAnnot* did for group cardinalities, the *GroupChild2LowerAnnot* creates an EAnnotation containing an OCL expression to check if the lower bound of a child cardinality is valid.

The relation states that for each Feature contained in a Group by means of a StructuralRelationship, a corresponding EAnnotation identified by the feature name must exist. As the `when` clause specifies, the rule is only applied if the corresponding EAnnotation has not been created yet. The `where` clause builds the OCL constraint. Such constraint specifies that the number of instances of the Feature must be higher than the lowerBound.

7.3.1.8 GroupChild2UpperAnnot *relation*

Fig. 7.12 shows the *GroupChild2UpperAnnot* relation. This relation is in charge of creating the EAnnotation to validate the number of

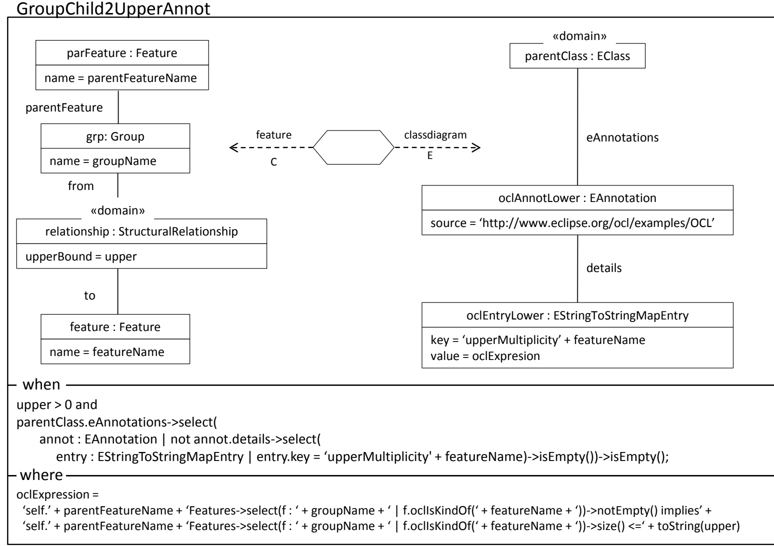


Figure 7.12: *GroupChild2UpperAnnot* relation

instances of a child Feature in a feature Group. It is specified in the same way than the *GroupChild2LowerAnnot* relation.

7.3.2 *Constraints over the DVM*

The *restriction relationships* and *model constraints* (FMCL expressions) are mapped to references and OCL expressions in the DVM. The following paragraphs summarize how these constraints are represented in the DVM.

First, the restriction relationships (implies, excludes, etc.) can be expressed as OCL expressions. The semantics of these relationships can be expressed as:

- `A implies B` relationship ($A \longrightarrow B$):

```
1 context PackageName inv:
2 A.allInstances()->notEmpty() implies B.allInstances()
3          ->notEmpty())
```

For example, the implies relationship that relates the features Automatic and TCS in Fig. 7.3 can be expressed as:

```
1 context CarPackage inv:
2 Automatic.allInstances()->notEmpty() implies
3         TCS.allInstances()->notEmpty())
```

- `A if and only if B` relationship ($A \longleftrightarrow B$):

```
1 context PackageName inv:
2 A.allInstances()->notEmpty() implies B.allInstances()
3         ->notEmpty()) and
4 B.allInstances()->notEmpty() implies A.allInstances()
5         ->notEmpty())
```

- `A excludes B` relationship (A ⤫—⤫ B):

```
1 context PackageName inv:
2 A.allInstances()->notEmpty() implies B.allInstances()
3         ->isEmpty()) and
4 B.allInstances()->notEmpty() implies A.allInstances()
5         ->isEmpty())
```

Second, for each Uses relationship between two Features, an EReference will be created in the target model. This EReference will relate two EClasses whose names will match the Features names.

Finally, the FMCL expressions are mapped to OCL expressions taking into account the mappings explained in section 7.3.1. Fig. 7.3 shows an example of this. As can be seen on the constraint that applies to the Wheel feature, a FMCL expression can be expressed directly using the OCL syntax. This way, an FMCL expression is directly transformed to an OCL invariant. The context of the invariant corresponds to the name of the *ConstrainableElement* that is linked to the constraint (dashed line in the figure) and the text of the expression remains the same:

The FMCL language can be considered as OCL enriched with syntactic sugar. This way, transforming a FMCL expression to an OCL expression is an straightforward process.

```
1 context Wheel
2 inv: self.radius > 15
3 inv: Wheel.allInstances()->forAll(w1, w2 |
4                             w1 <> w2 implies w1.radius = w2.
                                   radius)
```

However, although the FMCL expressions are almost the same than an OCL invariant, some simple conventions have been adopted to make the definition of model constraints closer to the feature modeling context. The semantics of these additions are defined by means of transformation patterns (see section 7.3.2.6).

An example of the application of some of these patterns can be seen in the constraint attached to the TCS feature (Fig. 7.3). The example constraint is transformed to the following OCL expression:

```
1 context TCS inv:
2 TCS.allInstances()->notEmpty() implies Engine.allInstances()
      ->forAll(power > 70000)
```

Next, the QVT–Relations rules which perform the transformation are explained in detail.

7.3.2.1 ExcludesRelationship2ModelConstraint *relation*

The top-level *ExcludesRelationship2ModelConstraint* relation (Fig. 7.13) is in charge of creating the OCL expression to guarantee that the existence of a feature excludes the existence of another feature. Thus, for each Excludes relationship contained in a FeatureModel,

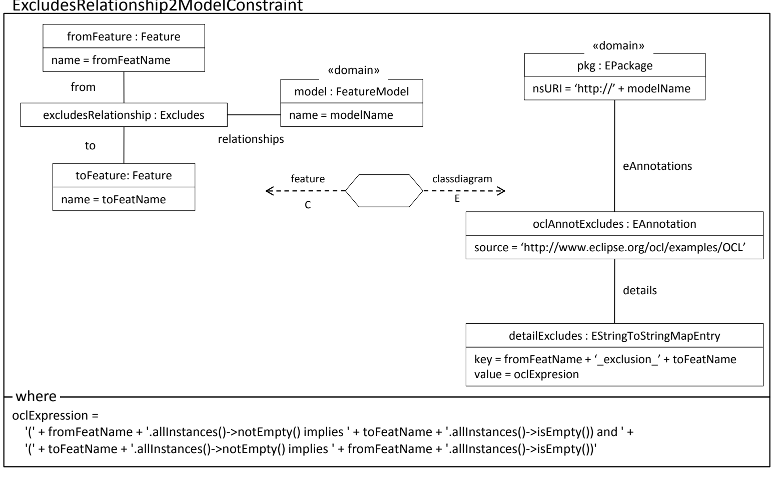


Figure 7.13: *ExcludesRelationship2Modelconstraint* relation

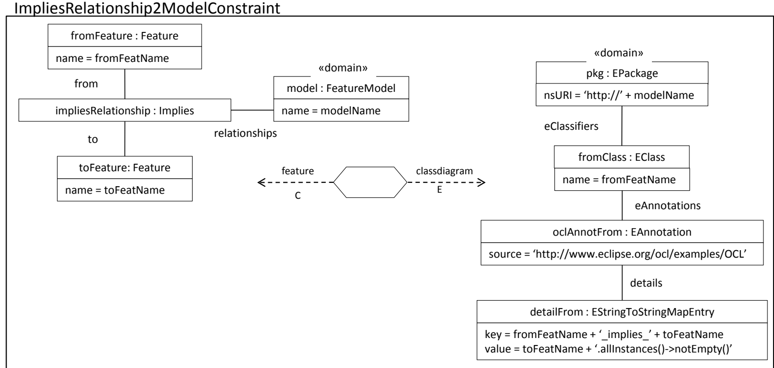


Figure 7.14: *ImpliesRelationship2ModelConstraint* relation

an EAnnotation is created in the corresponding EPackage. Such EAnnotation will define a key (an identifier) and a value (an OCL expression). Such attributes get their values from the Features related by the Excludes relationship (i. e., fromFeature and toFeature). Finally, the OCL expression is built in the `where` clause. It states that if the set of instances of the fromFeature feature is not empty, the set of instances of the toFeature feature must be empty and vice versa.

7.3.2.2 ImpliesRelationship2ModelConstraint *relation*

Fig. 7.14 shows the top-level relation *ImpliesRelationship2ModelConstraint*. This relation creates an EAnnotation to check that, when a feature has been selected, another feature has been selected too. In terms of the DVM this is achieved by checking that, when an EClass has at least one instance, the second EClass has at least one instance too.

In page 100, the OCL invariant defines the EPackage as the context of the constraint. However, this expression can be simplified if the context is set to the fromClass (the fromClass is the EClass which corresponds to the fromFeature). This way, the relation defines that for each Implies relationship, contained in a FeatureModel, and between two features (fromFeature and toFeature), an EAnnotation contained in the fromClass must exist. Such EAnnotation will con-

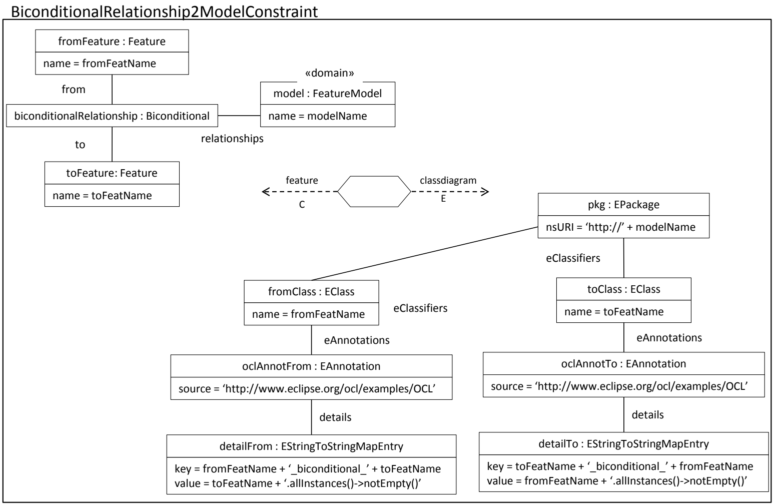


Figure 7.15: *BiconditionalRelationship2ModelConstraint* relation

tain a key and a value. The value will contain an OCL expression to check that the population of the EClass corresponding to toFeature must be not empty.

7.3.2.3 BiconditionalRelationship2ModelConstraint *relation*

The top-level relation *BiconditionalRelationship2ModelConstraint* is shown in Fig. 7.15. When a Biconditional relationship is defined between two Features, this relation is in charge of creating the EAnnotation to check that both (or none) of them have been instantiated.

The rule is written in a similar way to the *ImpliesRelationship2ModelConstraint*, as it expresses the same restriction but in both directions.

7.3.2.4 UsesRelationship2Reference *relation*

As opposed to the previous relationships which define restrictions at type level (regardless of the actual instance/s), the proposed Uses relationship defines relationships at instance level. I. e., once the Uses relationship is defined between two features (e. g., features A and B),

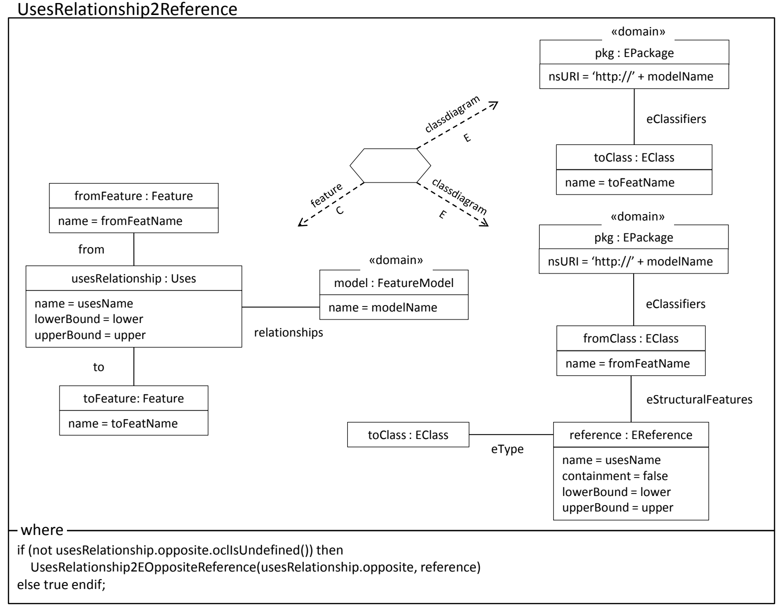


Figure 7.16: *UsesRelationship2Reference* relation

it can be used at configuration time to relate a specific instance of feature A with a specific instance of feature B.

The top-level relation *UsesRelationship2Reference* transforms the Uses relationship to an EReference, as EReferences are able to represent links among the instances of EClasses. Both the Uses relationships and the EReference are directed. In the source domain, a Uses relationships links a fromFeature with a toFeature. Such link has a name, a lower bound and an upper bound.

For this information, the relation creates an EReference (which is an eStructuralFeature of the fromClass) pointing to the toClass. Both the fromClass and the toClass are identified by their corresponding names.

The Uses relationship is directed and unidirectional. If the link needs to be navigable in both ways, the Uses relationship must be also defined in the opposite direction. This way, if it is defined in both

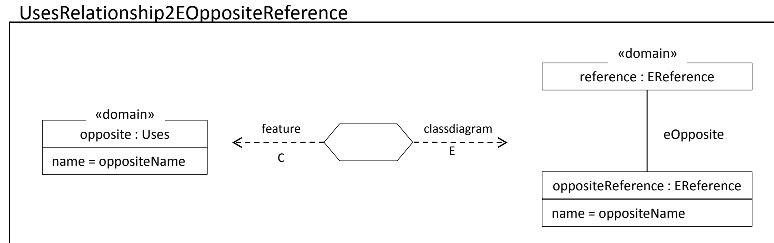


Figure 7.17: *UsesRelationship2EOppositeEreference* relation

ways, both relationships must be linked by means of the opposite role.

The `where` clause is used to invoke the *UsesRelationship2EOppositeEReference* relation. This relation is in charge of creating the inverse EReference when the Uses relationship is bidirectional (it is defined in both directions).

7.3.2.5 UsesRelationship2EOppositeEreference *relation*

The *UsesRelationship2EOppositeEreference* relation (see Fig. 7.17) is executed as a post-condition of the *UsesRelationship2Reference* rule. It receives a Uses relationship (which is the opposite one to the Uses relationship created previously in *UsesRelationship2Reference*), and whose name is oppositeName. This way, the rule creates a new opposite EReference using the oppositeName value.

7.3.2.6 FMCLConstraint2OCLConstraint *relation*

FMCL expressions are transformed to OCL expressions in the DVM. In the end, FMCL expressions are OCL expressions plus some syntactic sugar. FMCL constraints are applied over a *ConstrainableElement*. In the class diagram domain this is equivalent to describe an OCL in a specific *context*. Moreover, FMCL expressions are transformed to OCL using the simple patterns shown in Table 7.2. Such patterns are:

FMCL EXPRESSION PATTERN	EQUIVALENT OCL DEFINITION
ConstrainableElement	*ConstrainableElement*. allInstances()
ConstrainableElement. *property op expression*	*ConstrainableElement*. allInstances()->forAll(*property op expression*)
ConstrainableElement. selected()	*ConstrainableElement*. allInstances()->notEmpty()
FeatureName.childs()	*FeatureName*Type.allInstances()

Table 7.2: Summary of transformation patterns from FMCL to OCL

1. Using the name of a *ConstrainableElement* directly in a FMCL expression is a shortcut for using the `allInstances()` operation over that *ConstrainableElement*.

2. Taking into account the previous pattern, if operations are applied to properties of *ConstrainableElements*, they are performed/checked for all the corresponding instances.

3. The `selected()` operation can be used to query if a model element has been selected in a configuration (i. e., there exist instances of it).

4. The operation `childs()` can be used to collect all the child instances of *grouped features*. For example, `Transmission.-childs()` returns all the instances of the Manual and Automatic features. The equivalent OCL expression is `TransmissionType.allInstances()`.

FMCL expressions can be automatically translated to OCL expressions by a set of patterns. This patterns can be applied using standard tools such as as regular expressions or simple string replace functions.

Taking this patterns into account, Fig. 7.18 describes the relation to transform a FMCL expression to an OCL expression. In this case, only constraints applied to Features are transformed.

For each set of constraints in the source model which are applied to a Feature, an EAnnotation attached to an EClass in the corresponding EPackage will be created. Both the Feature and the EClass

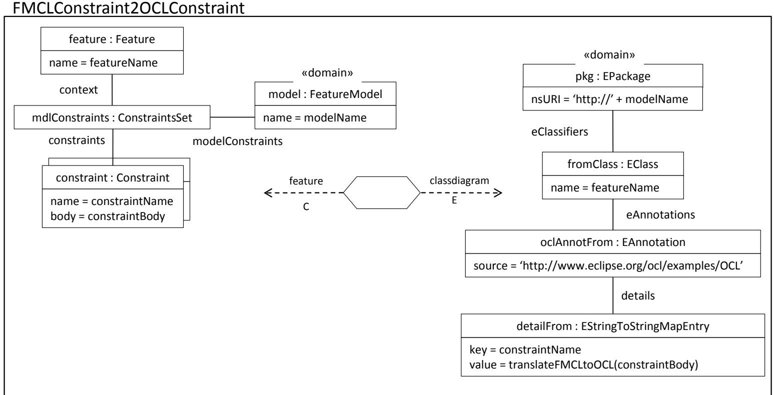


Figure 7.18: *FMCLConstraint2OCLConstraint* relation

must have the same name (featureName). A ConstraintSet contains a set of expressions (constraints) with a given name and body. Such expressions are transformed to the details of the created EAnnota-

Listing 7.3: *translateFMCLtoOCL(...)* OCL query

```
1  query translateFMCLtoOCL(expression : String) : String {
2    ConstrainableElement.allInstances()->iterate(
3      elt : ConstrainableElement;
4      s : String = expression |
5      -- The order when applying the substitutions is important
6      -- We must go from the most specific case to the most
          general one
7      if (elt.oclIsTypeOf(features::Feature)) then
8        s.replace('(' + elt.name + '\b)\.childs\(\)', '$1Type.
            allInstances()')
9      else
10       s
11     endif
12     .replace('(' + elt.name + '\b)\.(\w+\s+\S+\s+.+)', '$1.
          allInstances()->forAll($2)')
13     .replace('(' + elt.name + '\b)\.selected\(\)', '$1.
          allInstances()->notEmpty()')
14     .replace('(' + elt.name + '\b)(?:\.allInstances\(\))?', '
          $1.allInstances()')
15     )
16 }
```

tion. Finally, the body of the constraint is transformed to an OCL expression using the translateFMCLtoOCL helper function (Listing 7.3). Such function applies the transformation patterns defined in Table 7.2.

7.4 FEATURE MODEL CONFIGURATIONS

The main goal to transform a feature model to a class diagram is to define feature model configurations. Previous section describes how to convert a feature model to a DVM, which is in turn a class diagram. Fig. 7.19 shows the resulting model when the transformation is applied to the example feature model shown in Fig. 7.3.

In Fig. 7.19 annotations containing the OCL constraints are omitted for clarity purposes. The generated OCL expressions are shown in Listing 7.4.

Given this specification now it is possible to define new model configurations. Furthermore, it is possible to use any standard modeling tool to define model configurations and to check if they are valid or not. Fig. 7.20 shows a valid configuration for the example feature model. It represents a Car with four wheels. Each one of the wheels sizes 16 inches. The car is also equipped with an engine of 75,000 watts of power, automatic transmission and TCS.

Using DVMs a feature model configuration can be expressed as a classical object diagram.

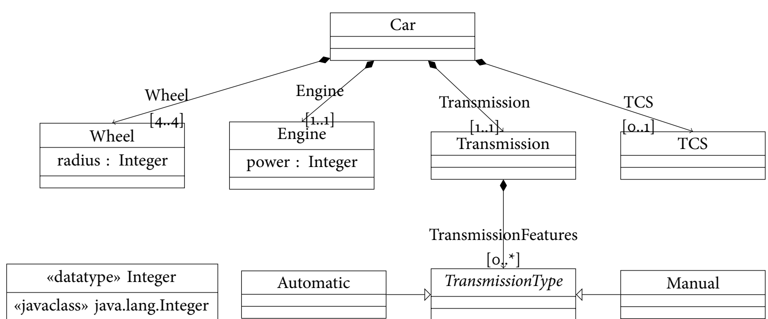


Figure 7.19: Generated class diagram for the example cardinality-based feature model

Listing 7.4: Generated OCL expression for the example feature model

```
1 package FeatureCar
2
3   context Wheel
4
5     inv radius_length: self.radius > 15
6
7     inv same_radius: Wheel.allInstances()
8           ->forAll(w1, w2 | w1 <> w2 implies w1.radius = w2.
               radius)
9
10
11  context TCS
12
13    inv power: TCS.allInstances()->notEmpty() implies
14          Engine.allInstances()->forAll(power > 70000)
15
16
17  context Automatic
18
19    inv Automatic_implies_TCS: TCS.allInstances()->notEmpty()
20
21
22  context Transmission
23
24    inv checkChildrenTransmission:
25    1 <= ( (if self.TransmissionFeatures->select(f :
         TransmissionType |
26              f.oclIsKindOf(Manual))->notEmpty() then 1 else
                   0 endif) +
27           (if self.TransmissionFeatures->select(f :
                TransmissionType |
28              f.oclIsKindOf(Automatic))->notEmpty() then 1
                   else 0 endif) + 0 )
29     and ( (if self.TransmissionFeatures->select(f :
          TransmissionType |
30              f.oclIsKindOf(Manual))->notEmpty() then 1 else
                   0 endif) +
31           (if self.TransmissionFeatures->select(f :
                TransmissionType |
32              f.oclIsKindOf(Automatic))->notEmpty() then 1
                   else 0 endif) + 0 ) <= 1
33
34    inv lowerMultiplicityManual:
35        self.TransmissionFeatures->select(f :
             TransmissionType |
36            f.oclIsKindOf(Manual))->notEmpty() implies
37         self.TransmissionFeatures->select(f :
              TransmissionType |
```

```
38              f.oclIsKindOf(Manual))->size() >=1
39
40      inv upperMultiplicityManual:
41          self.TransmissionFeatures->select(f :
                TransmissionType |
42              f.oclIsKindOf(Manual))->notEmpty() implies
43            self.TransmissionFeatures->select(f :
                  TransmissionType |
44              f.oclIsKindOf(Manual))->size() <=1
45
46      inv lowerMultiplicityAutomatic:
47          self.TransmissionFeatures->select(f :
                TransmissionType |
48              f.oclIsKindOf(Automatic))->notEmpty() implies
49            self.TransmissionFeatures->select(f :
                  TransmissionType |
50              f.oclIsKindOf(Automatic))->size() >=1
51
52      inv upperMultiplicityAutomatic:
53          self.TransmissionFeatures->select(f :
                TransmissionType |
54              f.oclIsKindOf(Automatic))->notEmpty() implies
55            self.TransmissionFeatures->select(f :
                  TransmissionType |
56              f.oclIsKindOf(Automatic))->size() <=1
57
58  endpackage
```

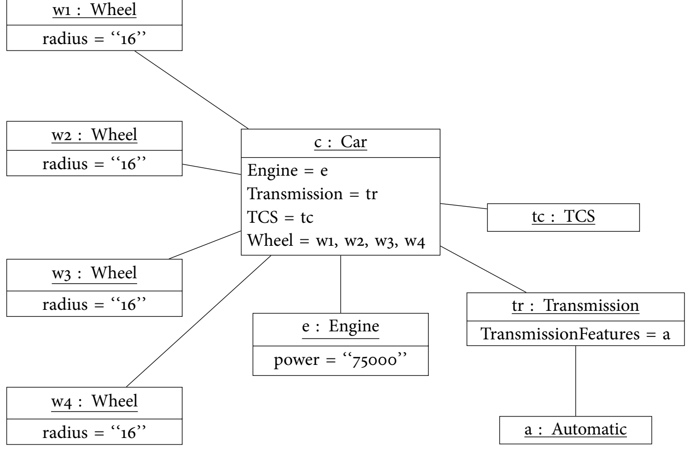


Figure 7.20: A valid configuration of the example DVM

7.5 SUMMARY AND CONCLUSIONS

In this chapter we have presented how to define and use feature models in a MDE process. This proposal addresses two issues: first, the incapability of nowadays metamodeling tools to deal simultaneously with artifacts located in all the MOF layers; and second, the complexity to define model constraints in feature models where features can be cloned and can have attributes. These problems have been solved by transforming feature models to DVMs that can be instantiated and reused in future steps of the MDE process.

Our proposal has been designed following the MDE principles and a metamodel for cardinality-based feature modeling has been defined. Feature models are automatically transformed in DVMs that are used to define configurations of feature models. Our approach has proposes a simple infrastructure to build configurations. Configurations are actually instances of a feature model (expressed by means of the DVM), so we can take advantage of the standard modeling tools. As feature models are described by DVMs that can be instantiated, both models and configurations can be used in other MDE tasks. Having a clear separation between feature models and configuration eases the validation tasks as they can be performed by means of built-in languages. Finally, as the transformation between feature models and DVMs is performed automatically by means of a declarative language we can trace errors back from DVMs to feature models.

It is noteworthy to remark the importance of using feature models and configurations at different layers. In chapter 8.4 an example where this architecture is used to integrate feature models in a MDE process is shown. This work describes how a model transformation with multiple inputs (feature models and functional models) is used to generate a software architecture automatically.

Part IV

THE MULTIPLE FRAMEWORK:

TOOL ARCHITECTURE

MULTI-MODEL DRIVEN SOFTWARE PRODUCT LINES DEVELOPMENT AND ANALYSIS

SUMMARY

This part describes the implementation of the MULTIPLE framework from a practical point of view, presenting both the tool architecture and the applications to different case studies. First, in chapter 8 the MULTIPLE framework is presented. This framework is built on top of the Eclipse platform and allows to describe different views of a software systems. Additionally, it provides the necessary tools to exploit the system's variability view. Second, in chapter 9 we present the Baseline Oriented Modeling (BOM)–Lazy approach. In this case study we put in practice our approach to describe and integrate feature models in a MMDSPL which in turn is a complex MDE process. The BOM–Lazy approach in an enhancement of the SPL proposed by Cabello Espinosa (2008) to the development of expert systems. Finally, chapter 10 aims to present the MULTIPLE framework from an industrial point of view. This chapter evaluates the scalability of the approach, using MULTIPLE to represent and analyse a large scale feature model provided by a worldwide aircraft engine manufacturing company.

8 THE MULTIPLE FRAMEWORK

«There are no big problems,
there are just a lot of little problems»

— Henry Ford
Pioneer of the assembly-line production method, 1863–1947

MULTIPLE is a generic framework which eases the development of software systems by using a Model-Driven approach. This framework is built on top of the Eclipse platform, and uses the Eclipse Modeling Framework (EMF) as it underlying metamodeling subsystem. However, MULTIPLE is not only a tooling to define models and metamodels, but it also provides a set of additional tools and built-in metamodels.

As MULTIPLE uses EMF, it can be considered a MOF-compliant tool. Thus, it can be used to implement and integrate the variability management proposal described in previous sections. This way, MULTIPLE is not only the suitable framework to carry out classic MDD processes, but also to design, implement and analyse multi-model driven SPLs. This goal is achieved by providing the following functionality:

- An extendable modeling and metamodeling subsystem. Such subsystem is populated by default with different metamodels and DSLs. For example, MULTIPLE provides:
  - A metamodel to describe systems' variability by using rich feature models.
  - A metamodel to describe functional views of software systems by defining its functional modules and the relationships among them.
  - A metamodel to describe architectural descriptions of software systems, using concepts such as components, connectors, roles, ports, services, etc.
  - A metamodel to describe PRISMA architectural models. The advantage of this metamodel is the ability to automatically obtain executable systems.
- A transformations subsystem. This subsystem is able to execute QVT–Relations model transformations.
- A validation subsystem. This subsystem is able to perform both conformance-checking and model-checking operations.
- A standardized way to interchange data and metadata among tools.

Throughout the following chapter we will describe the architecture of the tool, and the different components that make up the the MULTIPLE framework will be presented.

8.1 SUBSYSTEMS AND COMPONENTS OVERVIEW

MULTIPLE *is an extendable framework which can be easily enriched with new features and components.*

As explained, the MULTIPLE framework is built on top of the Eclipse platform. This allows developers to easily extend the framework as it is implemented following the Open Services Gateway initiative (OSGi) standard (OSGi 2008). Fig. 8.1 shows a general view of the architecture of the MULTIPLE framework. The figure shows the

dependencies among the different bundles in a simplified way. Each bundle (or plugin) is represented by a component. White components represent Eclipse plugins, and black components represent external tools which do not follow the OSGi architecture. Next, the main elements that can be identified in the figure are explained:

ECLIPSE PLATFORM This element includes a subset of the most representative plugins provided by the Eclipse platform which are used by the MULTIPLE plugins. They provide the basic functionality to execute the runtime and orchestrate the execution of the different components.

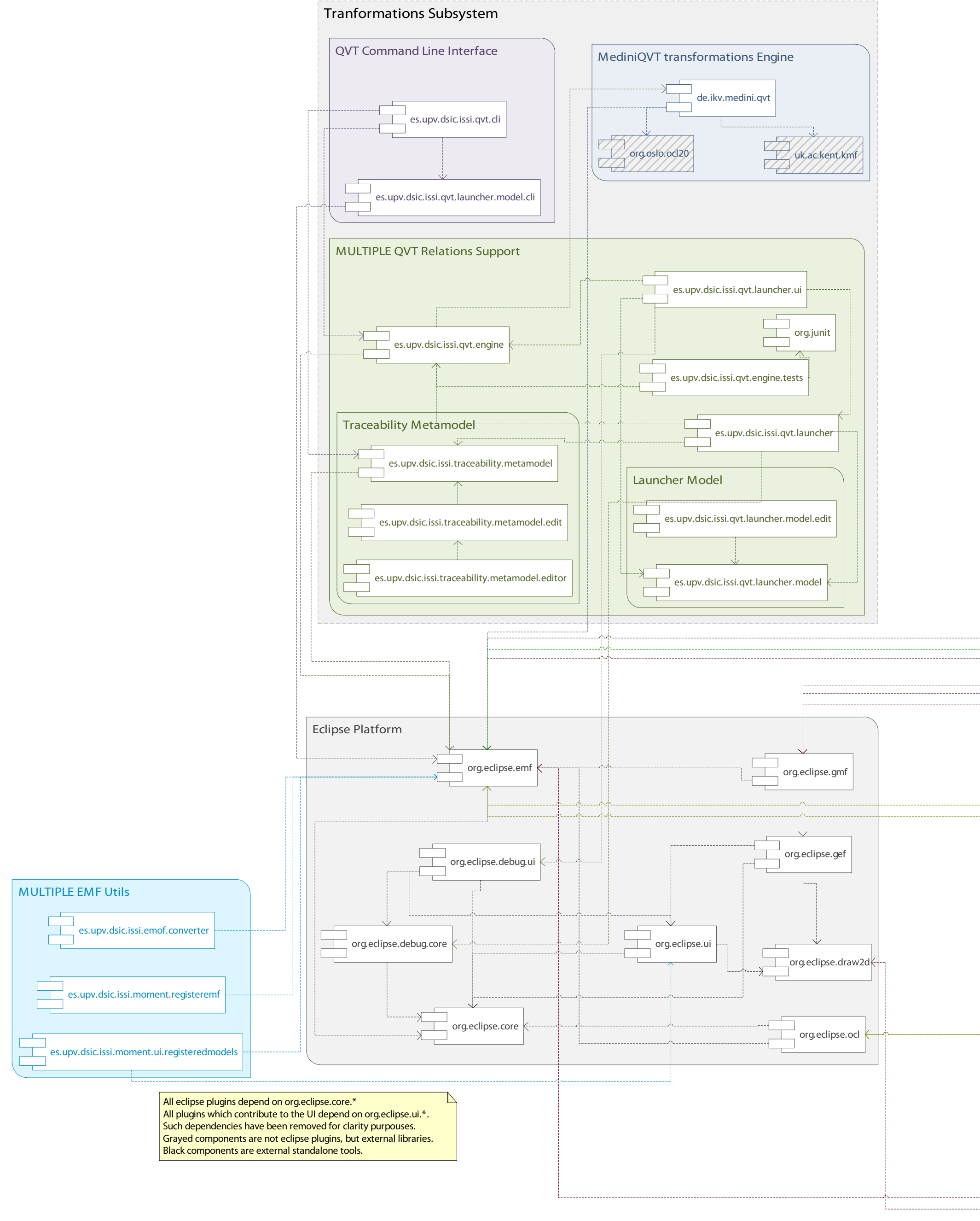
VARIABILITY METAMODEL SUPPORT This set of plugins implement the metamodel and the editors to define cardinality-based feature models according to the proposal made in chapter 7.

MODULAR METAMODEL SUPPORT This element provides the metamodel and the DSL to describe the functional view of a system by using modules, functions and dependency relationships

COMPONENT–CONNECTOR METAMODEL SUPPORT These components describe the metamodel and implement the graphical editor to define architectural models using a component–connector metaphor.

PRISMA METAMODEL SUPPORT These plugins implement the PRISMA metamodel. This allows the MULTIPLE framework to interoperate with the PRISMA-CASE and the PRISMA-MODEL-COMPILER tools.

TRANSFORMATIONS SUBSYSTEM This subsystem provides support to execute model transformations. It behaves as an interface between the user and the transformations engine. It is made up by the following sub-subsystems:


Tranformations Subsystem
QVT Command Line Interface
es.upv.dsic.issi.qvt.cli
es.upv.dsic.issi.qvt.launcher.model.cli
MediniQVT transformations Engine
de.ikv.medini.qvt
org.oslo.ocl20
uk.ac.kent.kmf
MULTIPLE QVT Relations Support
es.upv.dsic.issi.qvt.launcher.ui
es.upv.dsic.issi.qvt.engine
org.junit
es.upv.dsic.issi.qvt.engine.tests
Traceability Metamodel
es.upv.dsic.issi.qvt.launcher
es.upv.dsic.issi.traceability.metamodel
Launcher Model
es.upv.dsic.issi.traceability.metamodel.edit
es.upv.dsic.issi.qvt.launcher.model.edit
es.upv.dsic.issi.traceability.metamodel.editor
es.upv.dsic.issi.qvt.launcher.model
Eclipse Platform
org.eclipse.emf
org.eclipse.gmf
org.eclipse.debug.ui
org.eclipse.gef
MULTIPLE EMF Utils
es.upv.dsic.issi.emof.converter
org.eclipse.debug.core
org.eclipse.ui
org.eclipse.draw2d
es.upv.dsic.issi.moment.registeremf
org.eclipse.core
org.eclipse.ocl
es.upv.dsic.issi.moment.ui.registeredmodels
All eclipse plugins depend on org.eclipse.core.*
All plugins which contribute to the UI depend on org.eclipse.ui.*.
Such dependencies have been removed for clarity purpouses.
Grayed components are not eclipse plugins, but external libraries.
Black components are external standalone tools.

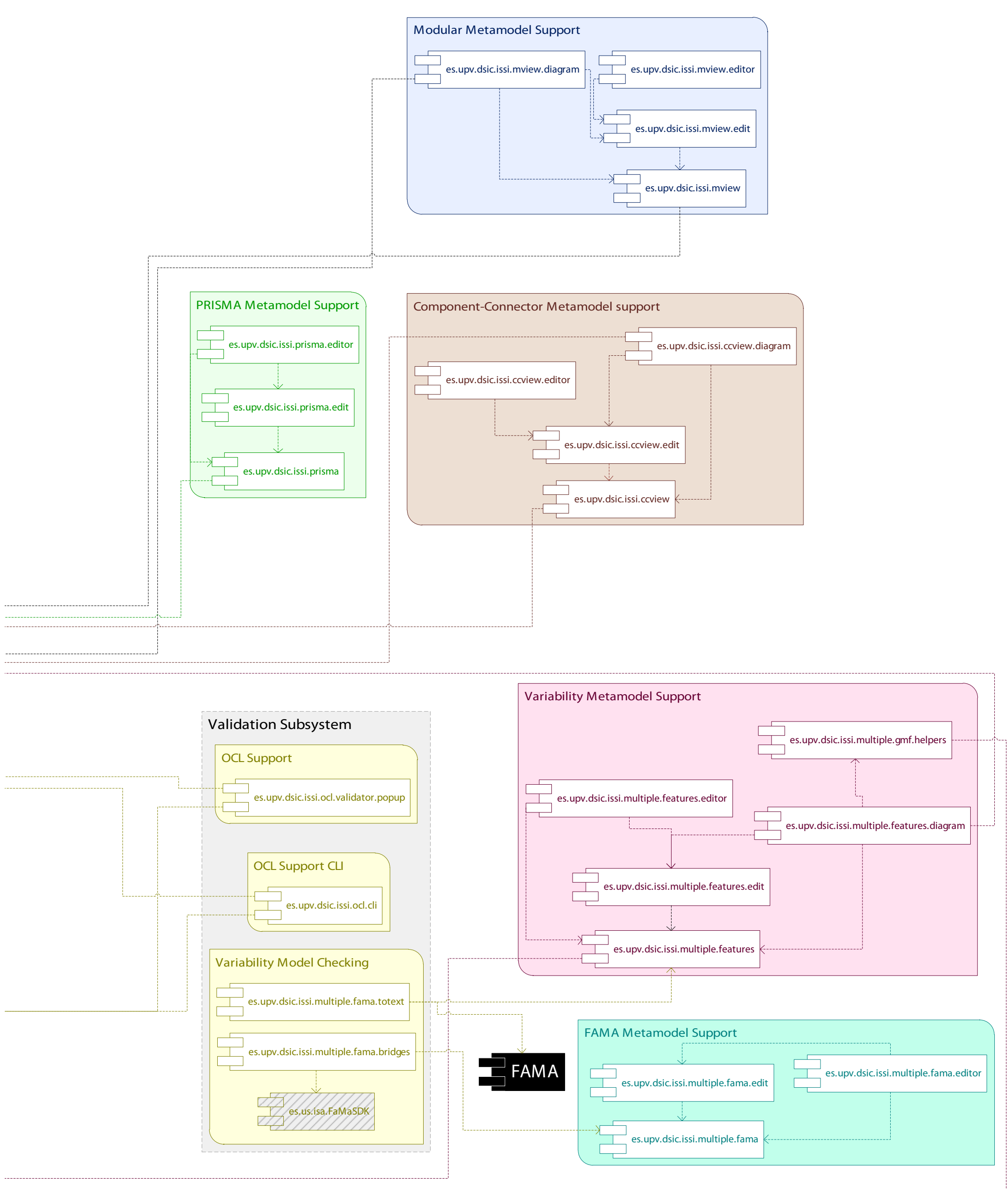


Figure 8.1: Architecture of the MULTIPLE framework

- MEDINIQVT TRANSFORMATIONS ENGINE — This sub-subsystem contains the implementation of the logic to execute MOF-compliant model transformations. It is based on the open source *medini QVT* engine.
- MULTIPLE QVT–RELATIONS SUPPORT — This sub-subsystem provides the infrastructure to execute model transformations inside the Eclipse platform. On the one hand, it provides the user interface to configure and execute model transformations. On the other hand, it provides the metamodel and the tools to the with traceability issues in a generic way.
- QVT COMMAND LINE INTERFACE — This sub-subsystem is built as a standalone java application to execute model transformations. It includes the plugins of the *Transformation subsystem* which do not contribute to the Eclipse User Interface (UI), and builds around them a textual interface to execute model transformations from a comand-line shell.

MULTIPLE *not only provides tools which are integrated into the* Eclipse *platform, but also standalone applications such as the* QVT command Line Interface *and the* OCL Command Line Interface.

VALIDATION SUBSYSTEM This subsystem provides support for validation tasks. Specifically, the *OCL Support* and *OCL Support CLI* elements provide support to check conformance relationships between EMF models (with OCL constraints) and EMF instances. They implement the user interface and communication mechanisms between user defined models and the internal EMF OCL engine. The former provides a graphical UI integrated in the Eclipse workbench, and the latter provides a command-line interface provided by a standalone application which packages the EMF OCL subsystem.

The *Variability Model Checking* component provides model checking capabilities. In this case, the model-checking capabilities are provided for models conformant to the MULTIPLE Variability Metamodel and the FAMA metamodel (see section 8.3.2). The *Variability Model Checking* component makes use

of the FAMA framework, an external tool which provides different solvers to validate feature models using formal methods.

MULTIPLE EMF UTILS This element encompasses some utility plugins, such as plugins to query and modify the metamodel registry (to dynamically load new metamodels), or a plugin to export models and metamodels using the standard XMI serialization format defined by MOF.

Next, the following sections explain each one of these subsystems and their components in deep detail.

8.2 THE ECLIPSE PLATFORM

As it has been pointed out previously in this thesis, Eclipse is an IDE which is built using a generic an extensible plugin system. The runtime of Eclipse is called Equinox, and it is an implementation of the OSGi standard (McAffer et al. 2009). All of the components that take part in the MULTIPLE framework are implemented as Eclipse plugins. The only exceptions to this are the tools which provide support to execute tasks using a command-line interface, which are regular Java applications. However, these standalone applications do already contain Eclipse plugins internally, and make use of their functionality (such as the EMF runtime).

Eclipse *has become the* de facto *standard tool to implement model-based solutions in the software engineering community thanks to its extensibility and its ecosystem of plugins and tools.*

Although the Fig. 8.1 has been simplified and only shows a small subset of the dependencies among plugins, it shows the most relevant ones. In the figure can be observed that all the plugins of the MULTIPLE framework depend on one or many core Eclipse plugins. For example, all Eclipse plugins depend on one or more plugins of the `org.eclipse.core` packages[1], as it provides the basic functionality to access the filesystem, job manager, etc.

Fig. 8.1 represents several basic components which are relevant to describe the MULTIPLE framework. These are:

1 Not all these dependencies have been drawn for clarity purposes.

org.eclipse.core — This component groups different plugins which provide basic functionality such as the `org.eclipse.core.runtime`, which provides the basic runtime to execute Eclipse plugins; the `org.eclipse.core.resources`, that allows plugins to access the files in the active *workspace* or the `org.eclipse.core.jobs`, which provide support to execute different tasks that can be monitored, among others.

org.eclipse.draw2d — This component provides the primitives to draw basic figures on screen. It requires the `org.eclipse.swt` library, which is the widget toolkit which interacts with the system's graphical API.

org.eclipse.ui — This component groups different plugins which implement the basic elements used to build the Eclipse UI. Any plugin which contributes to the Eclipse UI must include it among its dependencies.

org.eclipse.gef — This component implements the GEF framework. This is the basic framework used to create graphical editors for different diagrams.

org.eclipse.debug.core — This component provides support for running programs, breakpoint management, expression management, and debug events. It provides the basic extension points used by the QVT launcher component.

org.eclipse.debug.ui — This component (or plugin) provides the extension points to contribute the user interface elements to run external programs, builders, etc. It is used to implement the UI of the MULTIPLE QVT–Relations launcher.

org.eclipse.emf — This component groups all the plugins which implement the EMF framework. It is required by any plugin which deals with EMF models or metamodels. It provides the basic functionality to register models and metamodels, define instances, check basic conformance relationships, generate code, etc.

org.eclipse.gmf — This component implements the GMF framework. This framework combines the functionality of both the EMF and GEF frameworks. It allows developers to automatically generate the graphical editors which implement DSLs based on EMF models.

8.3 BUILT-IN METAMODELS

MULTIPLE provides by default some built-in metamodels to describe different views of systems. We have used MOF for the specification of these metamodels, as they can be easily implemented using *Ecore*. These metamodels are: the *variability metamodel*, the *FAMA metamodel*, the *modular metamodel*, the *component-connector metamodel* and the *PRISMA metamodel*. The plugins which implement these metamodels are explained next in a schematic way, as most of its code has been automatically generated by EMF.

MULTIPLE *implements several built-in metamodel to ease the description of different system views. These metamodels include those proposed by Limón Cordero (2010).*

8.3.1 *Variability metamodel support*

The built-in variability metamodel allows us to define feature models to manage the variability of systems. The components that take part in this subsystem implement the cardinality-based feature metamodel which was presented in section 7.2.

As it was shown in Fig. 7.2 (see page 88), the proposed metamodel allows us to define a variant of cardinality-based feature models. Such models can be enriched with feature attributes, cross-tree restrictions and complex model constraints using the FMCL language.

8.3.1.1 *Internal structure*

MULTIPLE provides different interfaces to manage variability in a proper way. Thus, different plugins which provide different functionality have been implemented. This way, feature models can be defined in three ways: (i) programatically in Java, using the generated API; (ii) using a simple tree editor; and (iii) using a graphical

editor which provides a DSL. Next the plugins which implement these interfaces are summarized.

ES.UPV.DSIC.ISSI.MULTIPLE.FEATURES This plugin implements the proposed metamodel to define cardinality-based feature models as an *Ecore* model. It is shown in Fig. 8.2 in the standard EMF tree editor. The core implementation of this plugin is automatically generated using the EMF code projector. The following plugins are required to run the `es.upv.dsic.issi.multiple.features` plugin:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.emf.ecore — The *Ecore* metamodel.

This plugin contains three packages as it is usual in the plugins generated by EMF:

features — This package contains the Java interfaces which are implemented by the classes contained in the next package (`feature.impl`). This way it is possible to simulate multiple inheritance in Java. EMF generates one Java interface for each one of the classes defined in the *Ecore* model.

features.impl — This package contains the classes that implement the proposed metamodel. This package also contains one class for each model class.

features.util — This package contains some utility classes.

The code generation patterns implemented by EMF are explained in (Steinberg et al. 2009) in depth.

ES.UPV.DSIC.ISSI.MULTIPLE.FEATURES.EDIT This plugin provides an intermediate layer between the user interface and the instances of the features metamodel. It implements the icon and label providers to customize how the model elements are shown to the user in the different model editors. For example, Table 8.1 shows

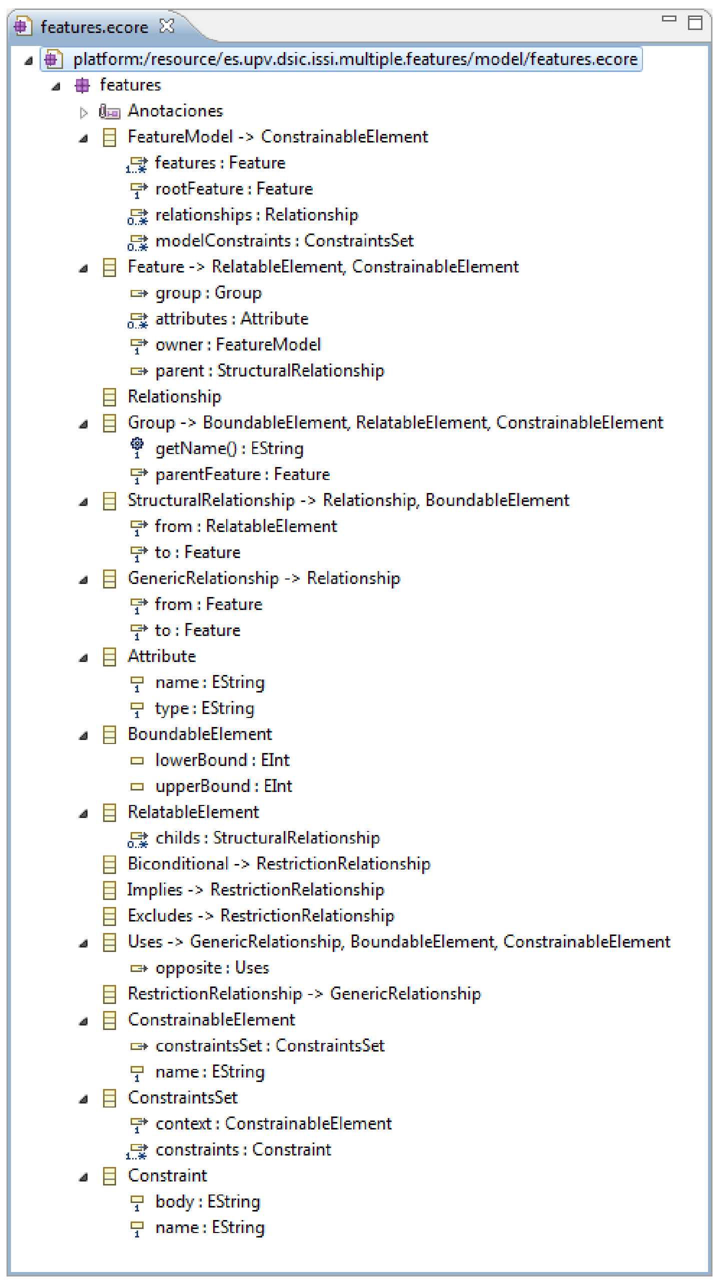


Figure 8.2: Features metamodel represented in an *Ecore* tree editor

the icons that are used to represent each one of the elements of the proposed features metamodel. It must be taken into account that different icons can represent the same element depending on the values of its attributes. For example, the mandatory relationship icon and the optional relationship icon are both used to represent structural relationships. The lower and upper bounds of the relationship will determine which one of the two icons should be used. If this can not be determined, the generic structural relationship icon will be used. The `es.upv.dsic.issi.multiple.features.edit` plugin has the following dependencies:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.emf.edit — The edit provider basic runtime.

es.upv.dsic.issi.multiple.features — The features metamodel.

The contents of this plugin are also automatically generated. This plugin only contains one package:

features.provider — This package contains the provider classes which determine how the model elements will be shown. The package contains one class for each class of the features metamodel. Some of the properties that can be customized are the label used to represent a model element, the icon of the element, the elements that should be considered as children of a given model element, the properties that can be modified in the *properties view* for a given element, etc.

ES.UPV.DSIC.ISSI.MULTIPLE.FEATURES.EDITOR This plugin implements a simple tree editor to define new feature model. The editor is the default editor for files with the **.features* extension. Moreover, the plugin provides a basic wizard to create new files of this type. The following plugin are required:

org.eclipse.core.runtime — The Eclipse basic runtime.

Table 8.1: Icons of the features metamodel elements

ICON	MODEL ELEMENT	ICON	MODEL ELEMENT
	Feature model		Feature
	Feature attribute		Structural relationship
	Mandatory relationship		Optional relationship
	Group		OR group
	XOR group		Implies relationship
	Biconditional relationship		Excludes relationship
	Uses relationship	{e ...}	Constraints set
{e}	Constraint		

org.eclipse.core.resources — The API to access workspace resources.

org.eclipse.emf.ecore.xmi — The plugin which provides support to the XMI persistence format.

org.eclipse.emf.edit.ui — The part of the *edit framework* which contributes to the Eclipse UI.

org.eclipse.ui.ide — The UI of the Eclipse IDE, adds support for error markers, input for file editros, etc.

es.upv.dsic.issi.multiple.features.edit — The different providers to represent features models in a proper way.

As it occurs with the previous plugins, the contents of this plugin are created by EMF. This plugin only contains one package:

features.presentation — This package contains the classes which implement the tree editor (`FeaturesEditor`), the new file wizard (`FeaturesModelWizard`) and the contribution to the Eclipse toolbar (`FeaturesActionBarContributor`).

ES.UPV.DSIC.ISSI.MULTIPLE.FEATURES.DIAGRAM This plugin implements the graphical editor to define our variant of cardinality-based feature models. The contents of this plugin have been created automatically by the GMF runtime. The source code of this plugin is determined by the different models that play a role in the GMF workflow (shown in Fig. 3.2, page 39). Fig. 8.3 how this workflow is configured for the features metamodel example. Specifically, this figure shows the *GMF Dashboard* view, which is the view used in Eclipse to guide the generation process until the source code is obtained. The most relevant models which determine the implementation of this plugin are the *gmfgraph* model, the *gmftool* model and the *gmfmap* model.

Fig. 8.4 shows the *gmftool* model. This model describes the palette of the graphical editor, i. e., which elements can be drawn, how are these elements grouped, and which are the icons which identify each one of these elements.

Fig. 8.5 shows what the *gmfgraph* model looks like. It defines the graphic primitives that should be used to draw the model elements. Such primitives are described in terms of basic figures (boxes, polygons, text, arrows, etc.). This model also specifies which of this graphic primitives will represent nodes, labels, links or containers.

Finally, Fig. 8.6 shows the most complex model: the *gmfmap* model. This model defines how the different elements of the previous

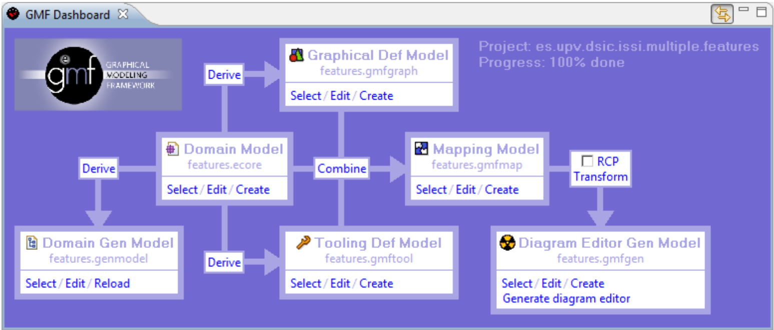


Figure 8.3: Workflow followed to obtain the features model editor

models interrelate. This way, the *gmfmap* model links each one of the elements of the feature metamodel with its corresponding graphical representation and tool. Complex constraints can be added to tune the graphical editor and ovoid invalid configurations.

ES.UPV.DSIC.ISSI.MULTIPLE.GMF.HELPERS This plugin implements some utility classes. The behaviour and appearance of GMF editors can not be fully customized in some cases using the source models. However, GMF allows to include custom classes which extend or modify the standard implementation. This plugin depends on the following plugins:

org.eclipse.draw2d — The basic drawing library.

org.eclipse.gmf.runtime.gef.ui — The GMF–GEF runtime library.

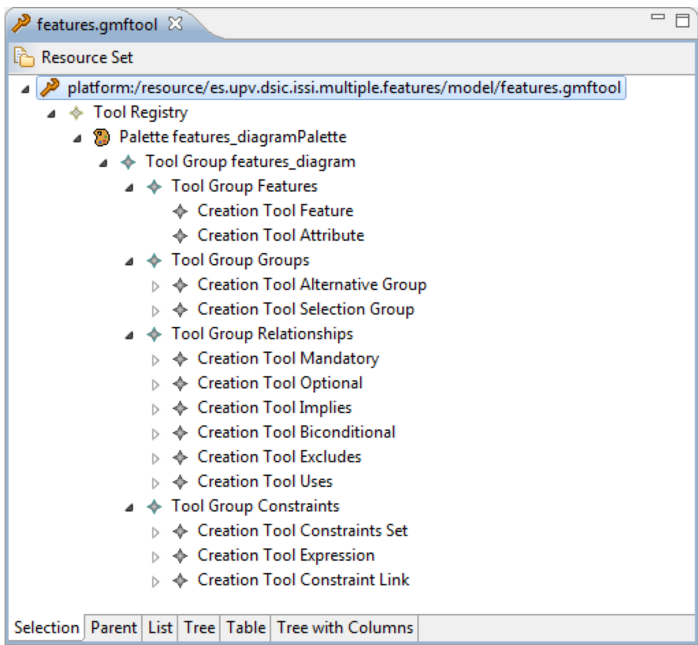


Figure 8.4: *Gmftool* model used to generate the GMF-based feature model editor

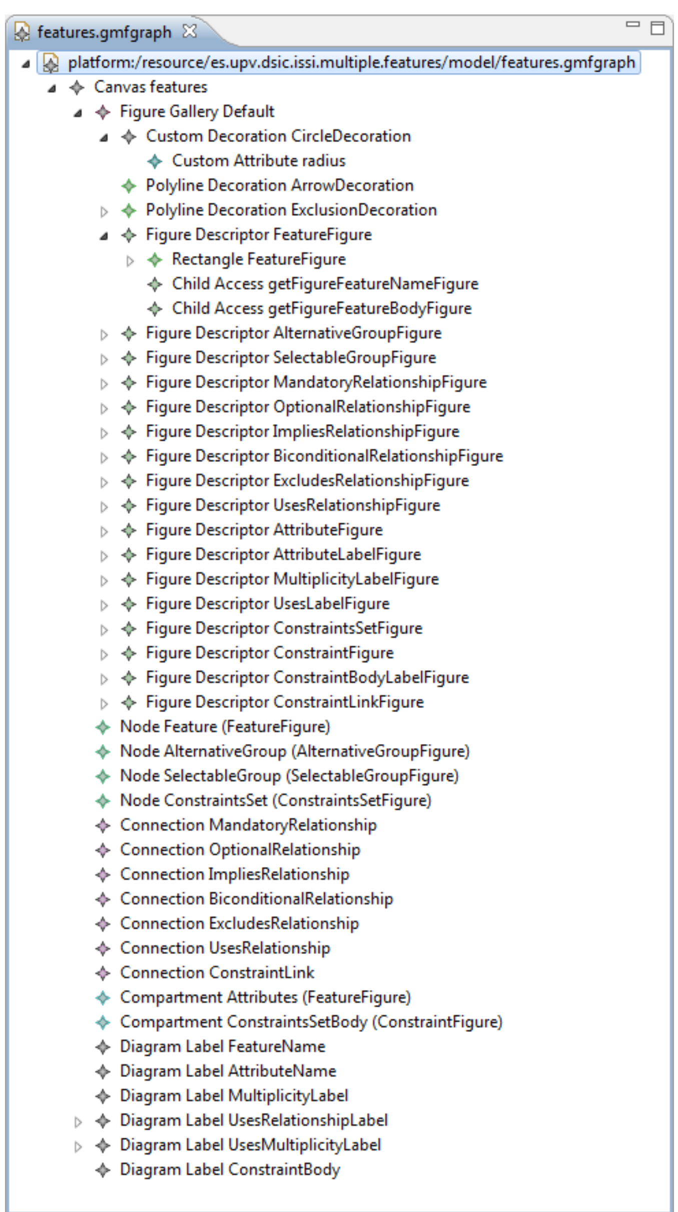


Figure 8.5: *Gmfgraph* model used to generate the GMF-based feature model editor

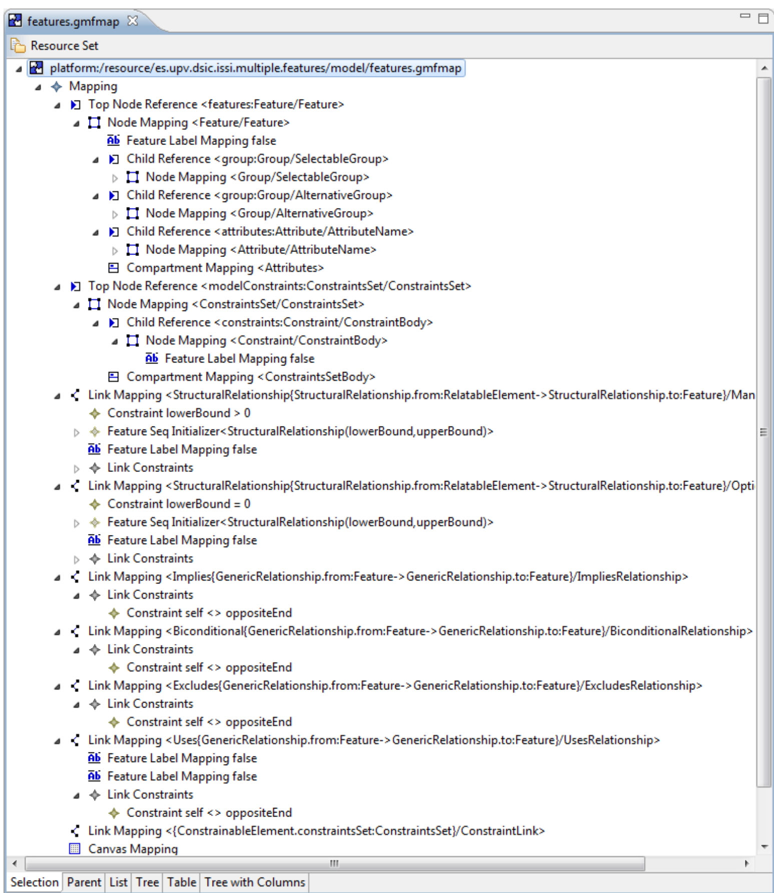


Figure 8.6: *Gmfmap* model used to generate the GMF-based feature model editor

All the utility classes are included in a single package, `features_diagram.diagram.edit.parts.helpers`. This packages includes the following classes:

FoldedRectangle — This class implements a custom figure, used to represent notes in the features model editor.

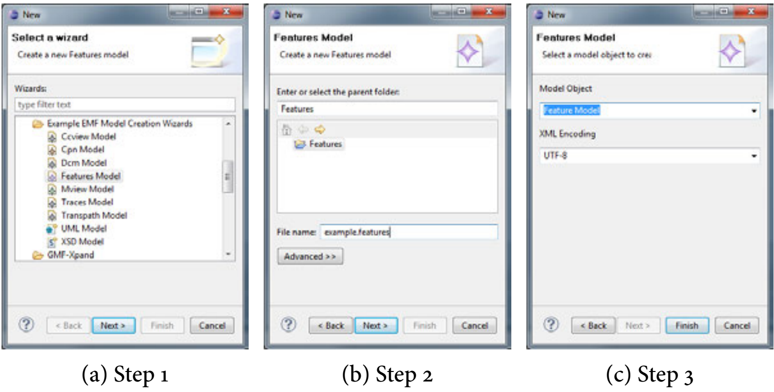
CircleDecoration — This class implements a circle decoration. This decoration is applied to arcs in the diagram to represent optional and mandatory features.

CustomSlidableAnchor, *MiddleSlidableAnchor* and *FixedConnectionAnchor* — These classes implement implement the logic that determines the anchoring points for arcs between different elements of the diagram.

8.3.1.2 *User Interface*

The feature modeling subsystem contributes different elements to the Eclipse user interface. Next, these elements are briefly presented.

NEW MODEL WIZARD The feature modeling component provides a wizard to easily create new feature models. Such wizard can be started from the standard dialog to create new elements in the workspace (File → New → Other...; CTRL + N), as shown in Fig. 8.7a. Next, a name must be issued for the new file that will be created in the selected folder of a project in the workspace (Fig. 8.7b). The new file must have the **.features* extension. Finally, as EMF artifacts have a tree structure, a root element must be selected. In the case of



(a) Step 1 (b) Step 2 (c) Step 3

Figure 8.7: New feature model wizard

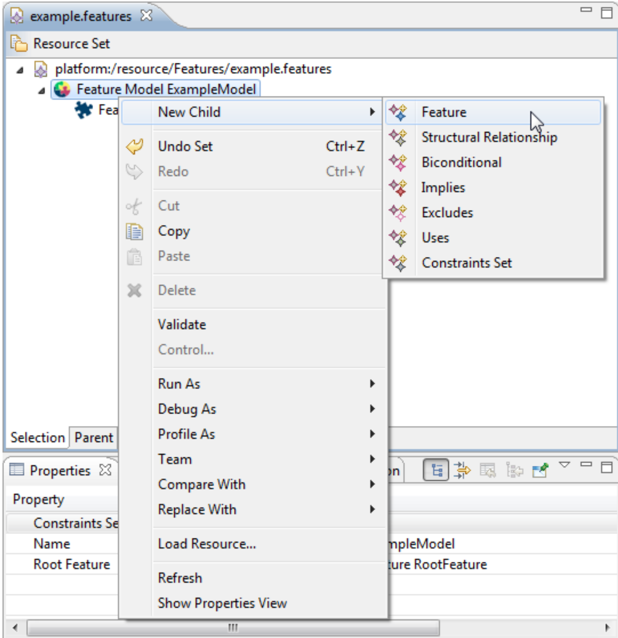


Figure 8.8: Standard tree editor for feature models

feature models the root element must be an instance of the *Feature Model* class, as show in Fig. 8.7c.

BASIC TREE EDITOR Once a new *.*features* file is created, it is automatically opened with the default tree editor. Fig. 8.8 shows this editor. It can be used to configure the feature model, adding new features, attributes, relationships or constraints. Fig. 8.8 shows a sample model, whose name is *Example model*. This model has only a feature, *Root feature*. The figure also shows how a new feature is created using the context menu, and using the "New Child → Feature" menu.

CREATING FEATURE MODEL DIAGRAM As can be seen, the tree editor is not a very user-friendly interface to deal with feature models. That is why a GMF-based graphical editor was developed.

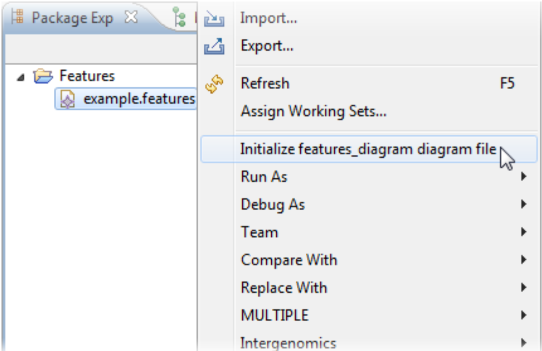
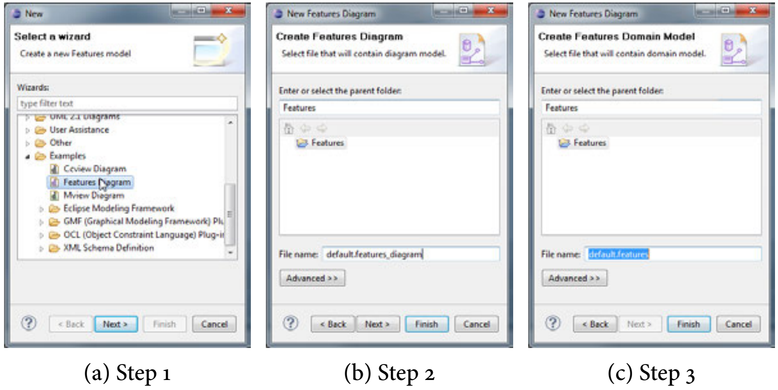


Figure 8.9: Initialize features diagram file

However, GMF-based editors need two files to store the needed information. First, they need the model file (i. e., the *.*features* file) and the diagram file. The diagram file stores the the information about the visualization of the model elements. For feature models, the diagram file has the *.*features_diagram* extension. To create a new diagram two choices are available: (i) we can initialize a new diagram for an existing feature model or (ii) we can create both the



(a) Step 1 (b) Step 2 (c) Step 3

Figure 8.10: New feature diagram wizard

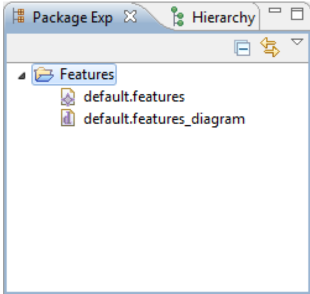


Figure 8.11: Example model and diagram files to represent a feature model graphically

feature model and the diagram from scratch. Fig. 8.9 shows how a new diagram can be created for an existing file.

To create both files from scratch a wizard is available. It can be launched from the *New* dialog (File → New → Other...; CTRL + N). Fig. 8.10 shows the different screens of the wizard. First, (step 1, Fig. 8.10a) the *New Feature Diagram* wizard must be select. Second, a name for the diagram file must be issued (step 2, Fig. 8.10b). The user can then go to step 3 (Fig. 8.10c) to provide the name of the model file and finish the wizard, or can finish the wizard at step two. In this case, the file name for the model will be automatically set.

Finally, Fig. 8.11 shows the two files that are needed by the graphical editor for feature models.

CARDINALITY-BASED FEATURE MODELING EDITOR As explained before, following the MDSD approach, graphical editors can be automatically generated from an EMF metamodel. As presented in the previous section, the `es.upv.dsic.issi.multiple.features.diagram` and `es.upv.dsic.issi.multiple.gmf.helpers` plugins implement the graphical editor for cardinality-based feature models using the traditional representation. This editor allows us to easily define new feature models.

Fig. 8.12 shows what this editor looks like. The palette is located on the right side of the figure, and shows the tools that can be used

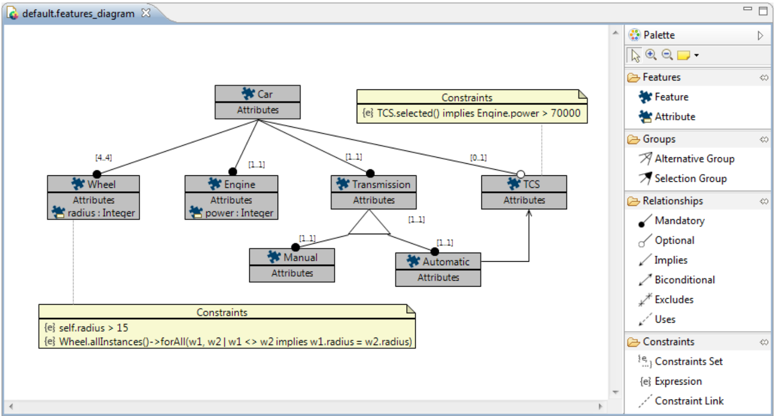


Figure 8.12: Example feature model

to define the feature models. In the canvas, the example feature model presented in Fig. 7.3 (page 90) is shown. As it was previously explained, the model describes a simple product line for cars. A car must have four wheels, one engine and a transmission. As an optional equipment the car can have a TCS. The figure also shows how constraints are represented: the arrow between the feature TCS and Automatic states that if an automatic transmission is selected, the TCS must be selected too; and the annotations attached to the TCS and Wheel features describe FMCL constraints.

8.3.2 *FAMA metamodel support*

The FAMA metamodel allows MULTIPLE to easily interchange models with the FAMA tool. This metamodel provides the basic infrastructure for the model checking capabilities included in MULTIPLE.

FAMA is a tool to analyse feature models. It provides different operations that can be applied to basic or extended feature models to guarantee if models are valid or not. These operations are implemented using different formalisms. This way, FAMA can assure that the results obtained from the application of such operations are logically correct. FAMA models can be basic FODA feature models or extended feature models. Extended feature models provide a limited support for attributes and model constraints, however the operations that are available for such kind of models are limited.

FAMA feature models can be represented as XML files and plain text files. To describe the structure of FAMA XML models a XSD description is provided within the FAMA tooling. This XSD (available in appendix E) can be used to generate an *Ecore* metamodel which allows to work with FAMA XML models natively in EMF. Next, the plugins that have been obtained to deal with FAMA models are explained.

8.3.2.1 *Internal structure*

The plugins that implement the FAMA metamodel are built using EMF, as the previously presented metamodel. However, the "*.ecore" file which represents the FAMA metamodel has been automatically obtained from a XSD file. This file describes the structure that FAMA models must respect. Fig. 8.13 shows the initial XSD file (8.13a), and the automatically obtained *Ecore* file (8.13b). Using this *Ecore* file, we are able to generate automaticalle the Java code to serialize and deserialize native FAMA XML files using EMF.

Although the FAMA metamodel support plugins have been obtained from an XSD file, in the end, the structure of the following plugins is similar to the structure of regular EMF plugins. Next, the plugins and summarized.

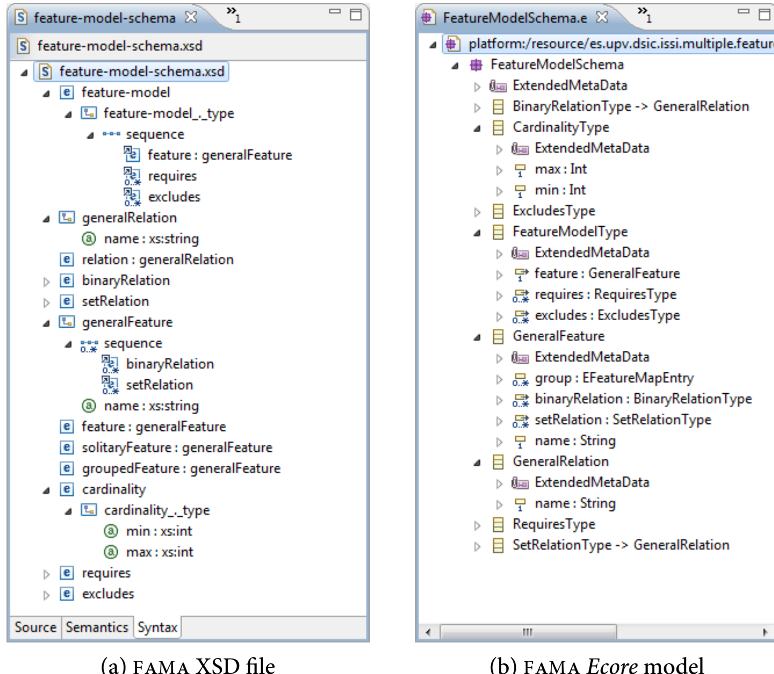
ES.UPV.DSIC.ISSI.MULTIPLE.FAMA This plugin implements the metamodel to describe FAMA models. The core implementation of this plugin is automatically generated by EMF, and requires the following plugins: `org.eclipse.core.runtime`, `org.eclipse.emf.ecore` and `org.eclipse.emf.xmi`.

This plugin contains three packages:

FeatureModelSchema — The interfaces package.

FeatureModelSchema.impl — The implementation package.

FeatureModelSchema.util — The utility package.



(a) FAMA XSD file (b) FAMA *Ecore* model

Figure 8.13: FAMA metamodel support initial files

ES.UPV.DSIC.ISSI.MULTIPLE.FAMA.EDIT This plugin provides the icon and label providers to customize how the model elements are shown to the user in the different model editors. The `es.upv.dsic.issi.multiple.fama.edit` plugin has the following dependencies:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.emf.edit — The edit provider basic runtime.

es.upv.dsic.issi.multiple.fama — The FAMA metamodel.

The contents of this plugin are packaged in a single package (`FeatureModelSchema.provider`), which is automatically generated.

ES.UPV.DSIC.ISSI.MULTIPLE.FAMA.EDITOR This plugin implements the simple tree editor and the needed wizards to build new FAMA models. The editor is the default editor for files with the **.fama* extension. The plugin has the following requirements:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.core.resources — The API to access workspace resources.

org.eclipse.emf.ecore.xmi — The plugin to support XMI.

org.eclipse.emf.edit.ui — The part of the *edit framework* which contributes to the Eclipse UI.

org.eclipse.ui.ide — The UI of the Eclipse IDE.

es.upv.dsic.issi.multiple.fama.edit — The different providers to represent FAMA models.

The contents of this plugin are created by EMF, and it only contains the `FeatureModelSchema.presentation` package.

8.3.2.2 *Equivalence between MULTIPLE feature models and FAMA feature models*

Cardinality-based feature models in MULTIPLE and FAMA models are almost equivalent, and it is quite straightforward to describe a set of equivalence relationships between the two. This way, a QVT–Relations transformation has been defined to stablish the relationships between both domains. This transformation is detailed in its textual representation in Appendix D. Table 8.2 summarizes the equivalences. As can be observed, the same set of basic primitives is shared among both domains (although using different names). Feature attributes are not considered, as analysis of attributed features models is very limited compared to traditional FODA feature models.

Next, the different relations are shown in its graphical representation toghether with a short explanation.

CARDINALITY-BASED FEATURE MODEL	FAMA FEATURE MODEL
FeatureModel	FeatureModelType
Feature	GeneralFeature
StructuralRelationship	BinaryRelationType
Group	SetRelationType
Implies	RequiresType
Excludes	ExcludesType
BoundableElement	Cardinality

Table 8.2: Correspondences between Cardinality-based feature models and FAMA

MODEL2MODEL RELATION The *Model2Model* relation states that, if a *FeatureModel* referencing a *Feature* (root) exists in the source domain; a *FeatureModelType* referencing a *GeneralFeature* (first) must exist in the target domain too. In this case, both features (root and first) must have the same name.

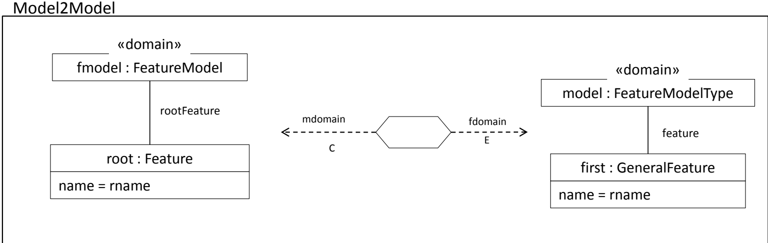


Figure 8.14: *Model2Model* relation

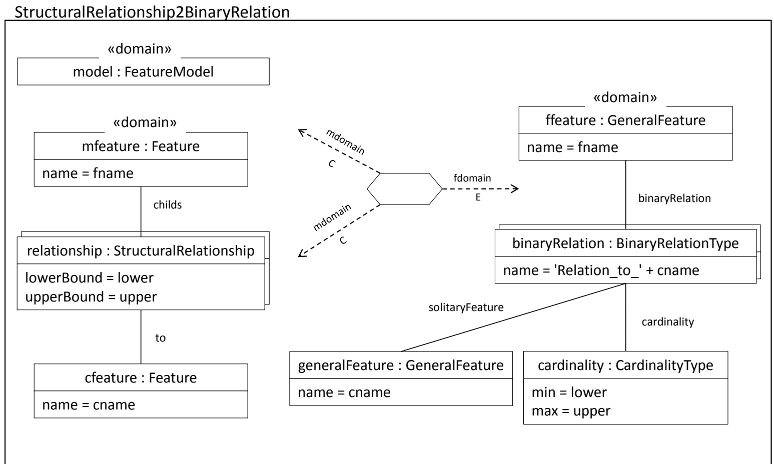


Figure 8.15: *StructuralRelationship2BinaryRelation* relation

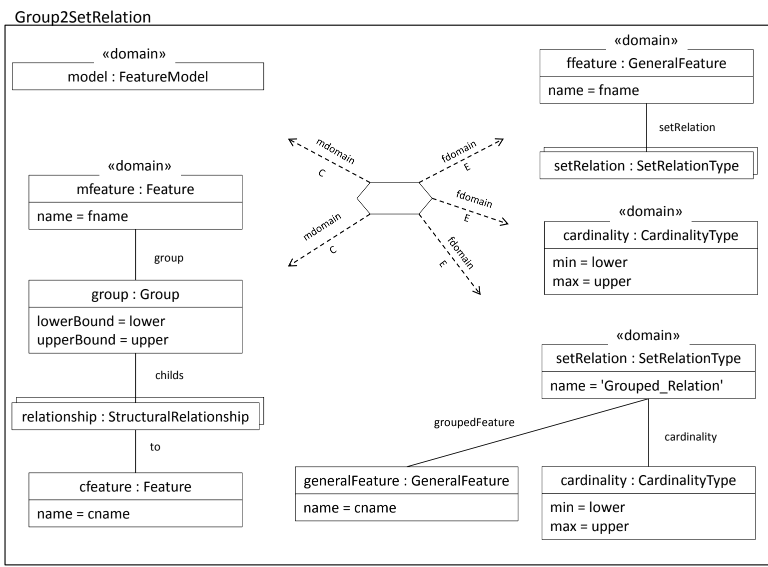


Figure 8.16: *Group2SetRelation* relation

STRUCTURALRELATIONSHIP2BINARYRELATION RELATION The *StructuralRelationship2BinaryRelation* relation describes the equivalence of parent-child relationships between both domains. In this case, if a *StructuralRelationship* is defined between two features in the source domain, a *BinaryRealtionType* relationship must be defined in the target domain. Parent features (*mfeature* and *ffeature*) must share the same name; and the same applies to the child features (*cfeature* and *generalFeature*). When the *BinaryRelationType* element is created, a *CardinalityType* element must also be created. This element stores the cardinality that the source *StructuralRelationship* defines.

GROUP2SETRELATION RELATION The *Group2SetRelation* relationship defines how the groups of features must be treated. This case is handled similarly to the previous one, when only parent-child features are considered. The only difference resides in the existence of the *Group* element in the source domain, which is mapped to a *SetRelationType* element in the target domain. The rest of the elements are mapped almost the same, i. e., parent features must share the same name, child features must also share the same name too, and the lower and upper bound of the source relationship are mapped to a *CardinalityType* element in the target domain.

EXCLUDESRELATIONSHIP2EXCLUDESTYPE RELATION The *ExcludesRelationship2ExcludesType* relationship describes how the excludes relationships must be transformed. In this case, given an *Excludes* relationship in the source domain, an *ExcludesTypeElement* must be created in the target domain. The name of the features that the *Excludes* element relates (*fromFeature* and *toFeature*) are used to fill the *excludes* and *feature* attributes of the *ExcludesType* element.

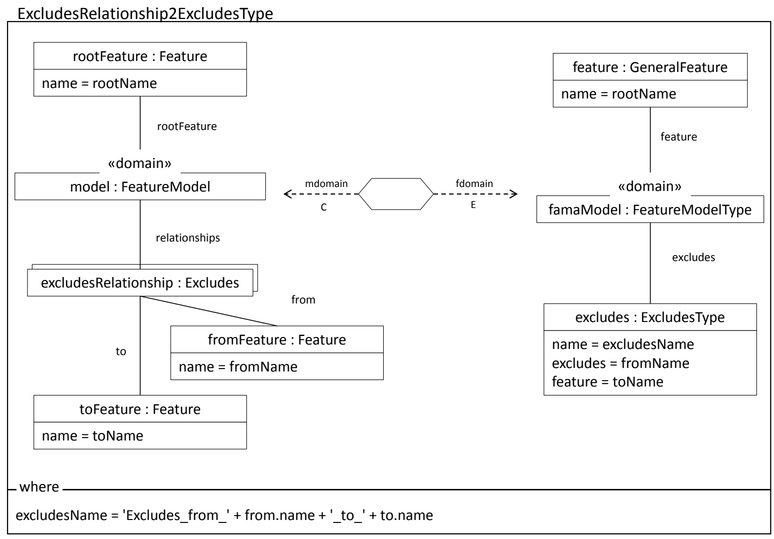


Figure 8.17: *ExcludesRelationship2ExcludesType* relation

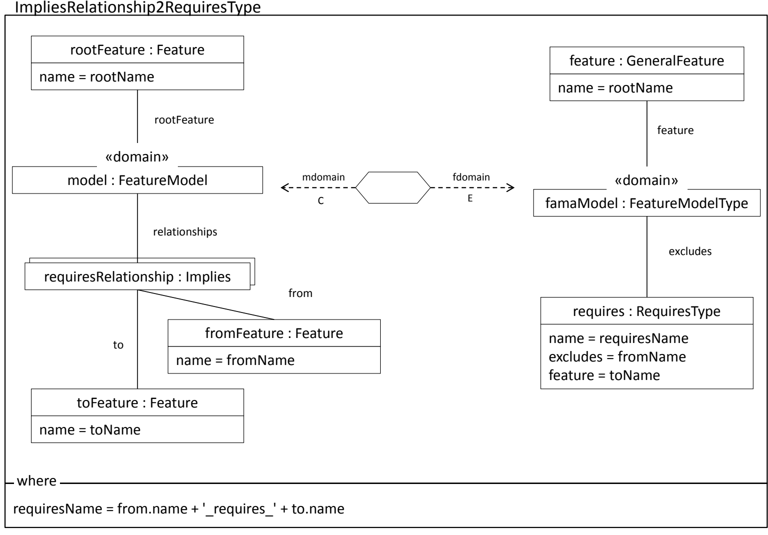


Figure 8.18: ImpliesRelationship2RequiresType relation

IMPLIESRELATIONSHIP2REQUIRESTYPE RELATION The *ImpliesRelationship2RequiresType* relationships defines how to transform an *Implies* relationship to a *Requires* relationship. As the *Implies* relationships are defined in a similar way than the *Excludes* relationships, this rule is almost equal to the *ExcludesRelationship2ExcludesType* rule. In this case, the mapping is established between an Implies and a *RequiresType* element.

8.3.3 *Modular metamodel support*

The modular metamodel implemented in MULTIPLE *is a simple functional model which implements the modular view proposed by Limón Cordero (2010).*

For the specification of the modular view metamodel several proposals have been analyzed. MULTIPLE provides support to specify both the modular and the component–connector view of a system as proposed in (Shaw and Clements 2006). In this sense, Limón Cordero (2010) proposes two metamodels to specify these views based on the relationships defined by Bass et al. (1998).

This way, Fig. 8.19 shows an implementation of the Modular View Metamodel (MM Modular view) as defined in (Limón Cordero 2010), using the class diagram representation of an *Ecore* model. The main element considered for this view is the module itself. This figure shows that a model contains a set of modules (which can contain different functions), which are linked to other modules by means of relations (decomposition, uses and layer). A module can be made up of several modules by using the decomposition relationship. This allows us to define hierarchical structures. By means of the Another relevant relation is the Use relation, which specifies relationships among modules. The labels in the links are useful for indicating how the relation is made. Finally, the Layer relationship allows to described layered models. The complete specification of the metamodel can be looked up in chapters 3 and 5 of (Limón Cordero 2010).

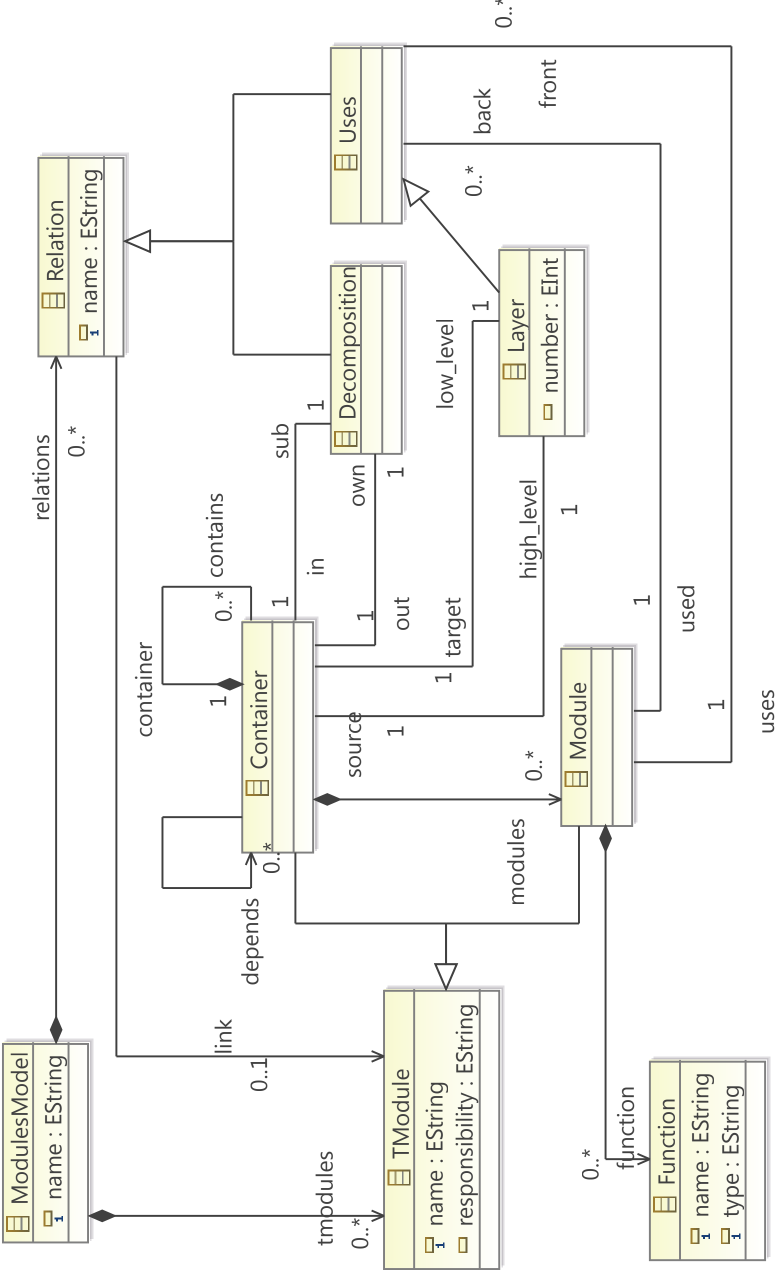


Figure 8.19: Modular view metamodel

8.3.3.1 *Internal structure*

The plugins which provide support to describe the system's modular view have been built using EMF and GMF. This way, the code of these plugins has been obtained almost entirely automatically. As a consequence, their internal structure is very similar to plugins presented previously. For this reason, only a short summary of the implementation details will be provided.

ES.UPV.DSIC.ISSI.MVIEW This plugin implements the metamodel to describe the modular view. The core implementation of this plugin is automatically generated by EMF, and requires the following plugins:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.emf.ecore — The *Ecore* metamodel.

This plugin contains three packages as usual:

es.upv.dsic.issi.mview — The interfaces package.

es.upv.dsic.issi.mview.impl — The implementation package.

es.upv.dsic.issi.mview.util — The utility package.

ES.UPV.DSIC.ISSI.MVIEW.EDIT This plugin provides the icon and label providers to customize how the model elements are shown to the user in the different model editors. The `es.upv.dsic.issi.mview.edit` plugin has the following dependencies:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.emf.edit — The edit provider basic runtime.

es.upv.dsic.issi.mview — The modular view metamodel.

The contents of this plugin are also automatically generated. This plugin only contains the `es.upv.dsic.issi.mview.provider` package.

ES.UPV.DSIC.ISSI.MVIEW.EDITOR This plugin implements the simple tree editor and the needed wizards to build modular models. The editor is the default editor for files with the **.mview* extension. The plugin has the following requirements:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.core.resources — The API to access workspace resources.

org.eclipse.emf.ecore.xmi — The plugin to support XMI.

org.eclipse.emf.edit.ui — The part of the *edit framework* which contributes to the Eclipse UI.

org.eclipse.ui.ide — The UI of the Eclipse IDE.

es.upv.dsic.issi.mview.edit — The different providers to represent modular models.

The contents of this plugin are created by EMF, and it only contains the `es.upv.dsic.issi.mview.presentation` package.

ES.UPV.DSIC.ISSI.MVIEW.DIAGRAM This plugin implements the graphical editor to define modular models. This plugin has been automatically created by the GMF runtime using the different models which define the editor (the *gmfgraph* model, the *gmftool* model and the *gmfmap* model).

Fig. 8.21 shows the *gmgraph* model, which specifies how the different elements should be represented. As can be observed in the figure, different graphical primitives are specified to represent *modules*, *uses relationships*, *function compartments* and *function labels*. Fig. 8.21 shows the elements that will appear in the palette of the canvas, and finally, Fig. 8.22 shows how the different models are interrelated. As can be seen, the main elements that can be dropped in the canvas are the modules, and the links that can be painted among them are *uses relationships*.

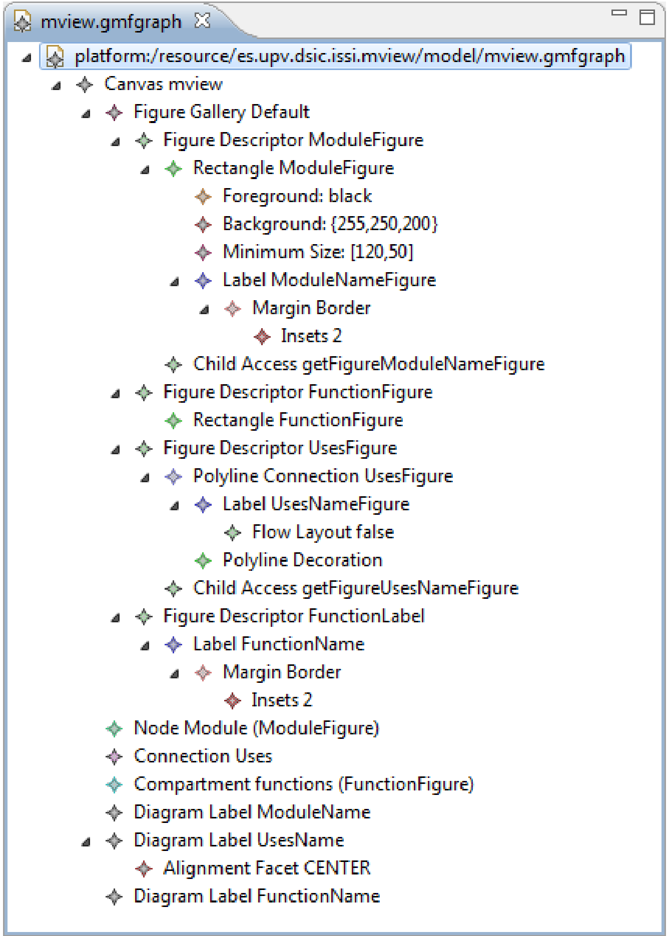


Figure 8.20: *Gmfgraph* model used to generate the GMF-based modular model editor

8.3.3.2 *User Interface*

The modular metamodel subsystem contributes different menus, wizards and editors to the Eclipse interface. These contributions are automatically generated by EMF or GMF, such as the new model wizard or the tree editor.

STANDARD CONTRIBUTIONS The modular view metamodel provides almost the same contributions to the Eclipse UI than the features metamodel. Some of these contributions are EMF-dependent, such as the *New modular model wizard* and the *Basic tree editor*; and

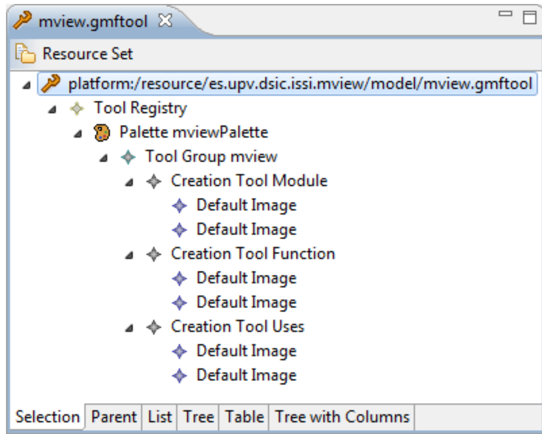


Figure 8.21: *Gmftool* model used to generate the GMF-based modular model editor

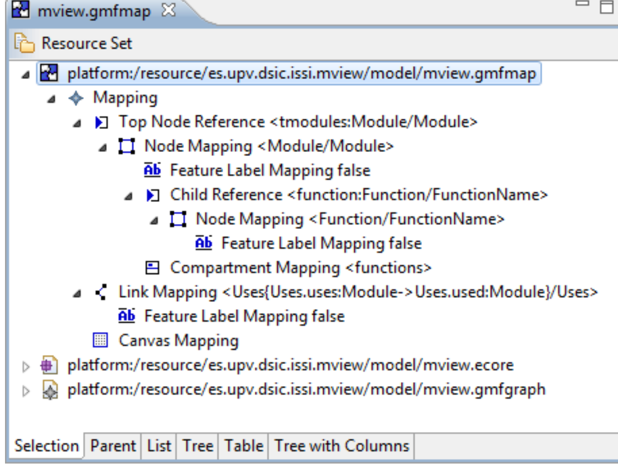
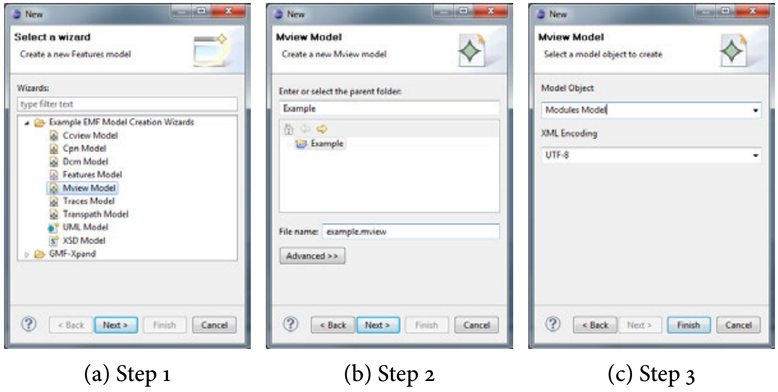


Figure 8.22: *Gmfmap* model used to generate the GMF-based modular model editor

others are GMF-dependent, such as the *New modular diagram editor* or the *Modular view editor*. Both the EMF and GMF-based generated contributions are very similar among different metamodels with only a few differences. For example, if we compare Fig. 8.23 (which shows the *New modular model wizard*) with Fig. 8.7 (which shows the *New features model wizard*) we observe that they look almost the same.

For this reason and to avoid redundancy, we will not show what the "standard UI contributions" look like from this moment on. We understand as "standard UI contributions" the following: the *new model wizard* (already shown in Figs. 8.7 and 8.23), the *standard tree editor* (shown in Fig. 8.8), the *new diagram wizard* (shown in Fig. 8.10) and the *initialize new diagram file* menu (shown in Fig. 8.9).

An exception to this convention will be applied to the GMF-based editors as they are metamodel-dependent. GMF-based editors implement a different graphical DSL for each metamodel, and although they share the same structure (canvas, palette, menus…), they express substantial differences in their actual contents.



(a) Step 1 (b) Step 2 (c) Step 3

Figure 8.23: New modular model wizard

MODULAR VIEW EDITOR The modular view editor is the component that provides a user-friendly interface to define new modular models. Built with the help of GMF, it implements a simple DSL that can be used to define modules, functions and simple use relationships.

Fig. 8.24 show a simple example which illustrates it. The editor is divided in two parts: the canvas (left) and the palette (right). The palette shows the tools that can be used to define new graphical elements. The elements that appear in the palette are directly related with the elements shown in the *gmftool* model (shown in Fig. 8.21). The editor also provides some context-sensitive creation tools. For example, the figure shows a small pop-up balloon which is shown when the mouse stands still on the canvas. This balloon contains a shortcut to the "Create new module" tool.

In the canvas a sample model which describes an electronic calculator (the system) is shown. This system is made up of three modules: the calculator module, the display, and the keyboard. The calculator module is in charge of performing arithmetic operations. It also has a memory which is able to store a floating point number. This

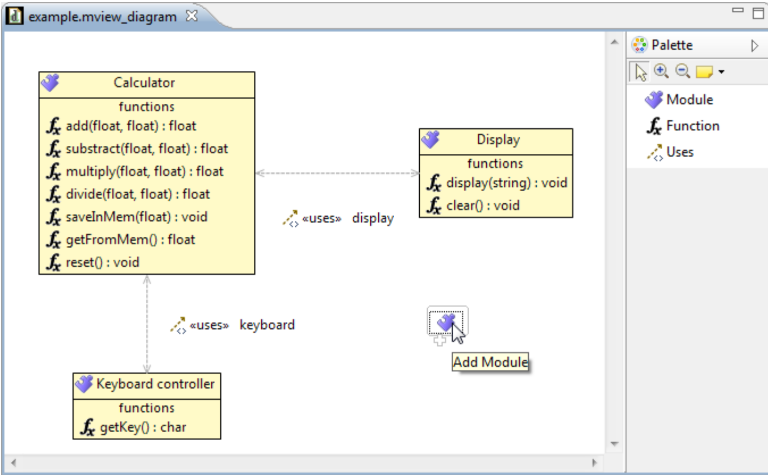


Figure 8.24: GMF-based modular view editor

memory can be accessed and cleared using the *saveInMem(...)* and *getFromMem(...)* functions. The calculator module makes use of the other two modules. The keyboard is managed by a controller module, that is used to track the keystrokes. Finally, the display module is in charge of showing the information to the user.

8.3.4 *Component–connector metamodel support*

The component–connector metamodel implemented in MULTIPLE *is a simple architectural model which implements the component–connector view proposed by Limón Cordero (2010).*

The component–connector metamodel has also been implemented based on the proposal made in (Limón Cordero 2010). Fig. 8.25 shows the Component-Connector View Metamodel. A model is made of a set of components and connectors, which are the main elements. Both are derived from a more general component class (TComponent). The components provide a set of services through a set of ports. The connectors link the ports of the components by means of their roles. Different types of *relations* can be also defined among *components* and *connectors*. The complete specification of the metamodel can be found in (Limón Cordero 2010).

8.3.4.1 *Internal structure*

The plugins that implement the component-connector metamodels are built using EMF and GMF, as the previously presented metamodels. This way, their structure is very similar to plugins presented previously. Next, the plugins and summarized.

ES.UPV.DSIC.ISSI.CCVIEW This plugin implements the metamodel to describe the component-connector view. The core implementation of this plugin is automatically generated by EMF, and requires the usual plugins (`org.eclipse.core.runtime` and `org.eclipse.emf.ecore`).

This plugin contains three packages:

es.upv.dsic.issi.ccview — The interfaces package.

es.upv.dsic.issi.ccview.impl — The implementation package.

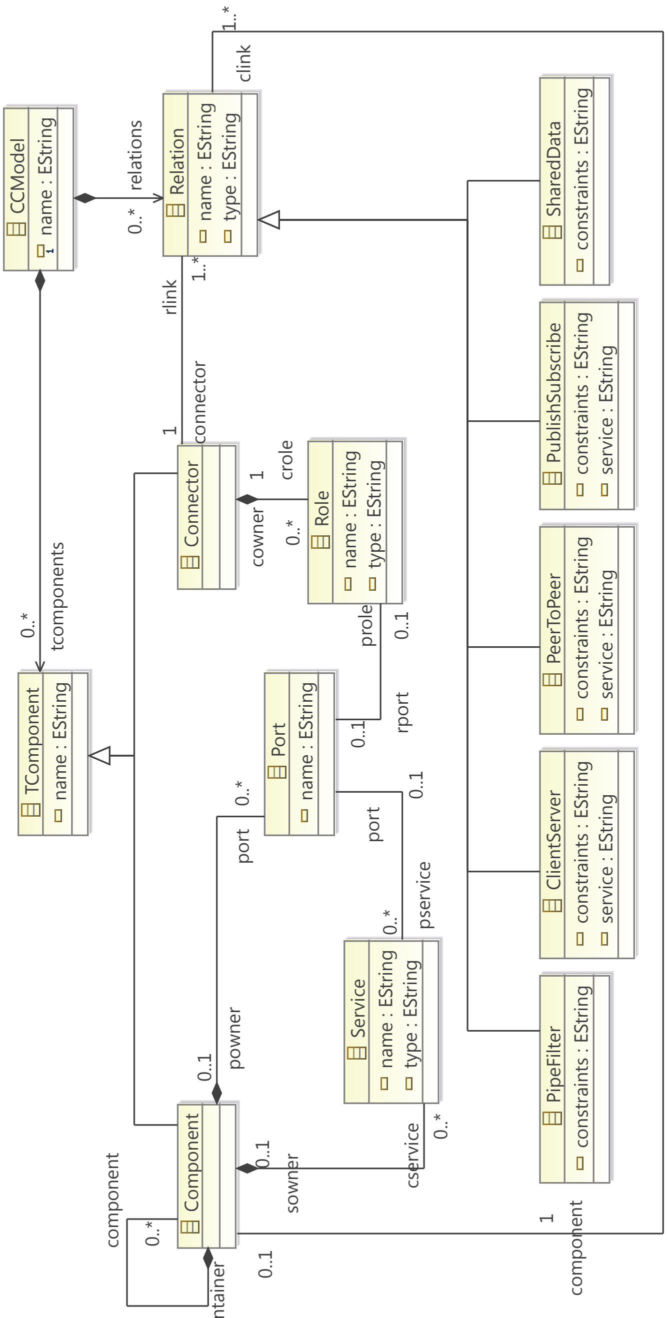


Figure 8.25: Component-connector view metamodel

es.upv.dsic.issi.ccview.util — The utility package.

ES.UPV.DSIC.ISSI.CCVIEW.EDIT This plugin provides the icon and label providers to customize how the model elements are shown to the user in the different model editors. The `es.upv.dsic.issi.mview.edit` plugin has the following dependencies:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.emf.edit — The edit provider basic runtime.

es.upv.dsic.issi.ccview — The modular view metamodel.

The contents of this plugin are packaged in a single package (`es.upv.dsic.issi.ccview.provider`), which is automatically generated.

ES.UPV.DSIC.ISSI.CCVIEW.EDITOR This plugin implements the simple tree editor and the needed wizards to build modular models. The editor is the default editor for files with the **.ccview* extension. The plugin has the following requirements:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.core.resources — The API to access workspace resources.

org.eclipse.emf.ecore.xmi — The plugin to support XMI.

org.eclipse.emf.edit.ui — The part of the *edit framework* which contributes to the Eclipse UI.

org.eclipse.ui.ide — The UI of the Eclipse IDE.

es.upv.dsic.issi.ccview.edit — The different providers to represent component-conector models.

The contents of this plugin are created by EMF, and it only contains the `es.upv.dsic.issi.ccview.presentation` package.

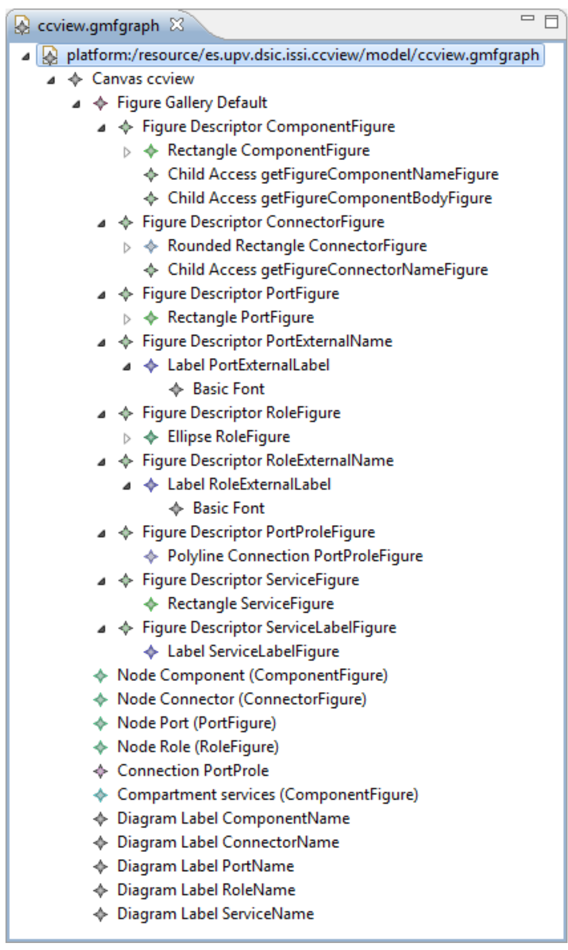


Figure 8.26: *Gmfgraph* model used to generate the GMF-based component-connector model editor

ES.UPV.DSIC.ISSI.CCVIEW.DIAGRAM This plugin implements the graphical editor to define component-connector architectural models. As in previous metamodels, this plugin has been automatically generated using the GMF runtime. Next, the different models which define the editor are shown.

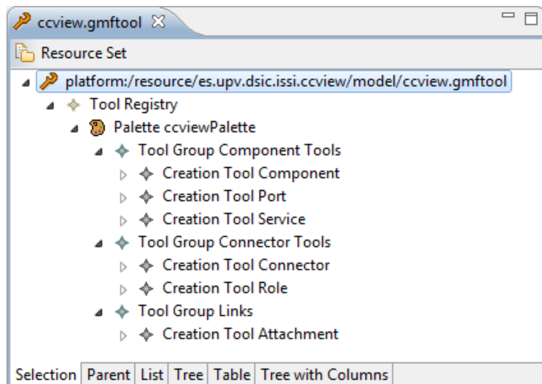


Figure 8.27: *Gmftool* model used to generate the GMF-based component-connector model editor

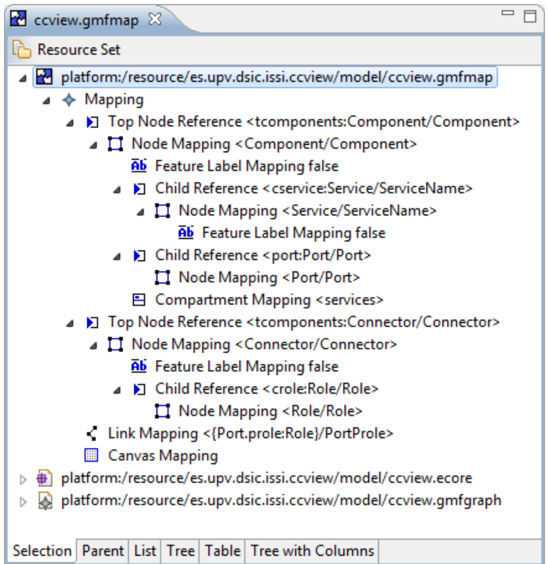


Figure 8.28: *Gmfmap* model used to generate the GMF-based component-connector model editor

Fig. 8.27 shows the *gmgraph* model. This model specifies the graphical primitives that will represent the domain model elements in the canvas. This way, *components* are drawn using a *rectangle*,

connectors are drawn using a *rounded rectangle*, ports are drawn using a small rectangle (the size of the rectangle is specified in the unfolded children of the PortFigure element), *roles* are drawn using an *ellipse*, etc.

Fig. 8.27 shows the elements that will appear in the palette of the canvas. As can be seen, elements are grouped in three categories *Component Tools*, *Connector Tools* and *Links*. The tools to create *Componentes*, *Ports* and *Services* are located in the first group; the tools to create *Connectors* and *Roles* are located in the second group; and finally, the tool to create the *Attachments* among *roles* and *ports* is located in the third group.

Finally, Fig. 8.28 shows how the different models GMF are interrelated with the domain model. The figure shows that the top nodes (the elements that are directly drawn in the canvas) are the *components* and the *connectors*. Moreover, for both elements a label is specified (which is used to represent the name of these elements). As can be seen, the *component nodes* can have both *services* and *ports*. The connector nodes can have roles. Finally, a *link mapping* is established between *roles* and *ports*.

8.3.4.2 *User Interface*

The component-connector metamodel contributes the same menus, wizards and editors to the Eclipse interface than the modular metamodel. These contributions are automatically generated by EMF or GMF. These contributions are the previously presented as the *standard contributions*, mainly: the *New component-connector model wizard*, the *Basic tree editor*, the *New component-connector diagram wizard* and the *Component-connector view editor*. As these types of contributions were presented previously for variability models and modular models, they will not be shown again.

COMPONENT-CONNECTOR VIEW EDITOR The component-connector view editor is the contribution which provides a user-friendly interface to define new component-connector architectural

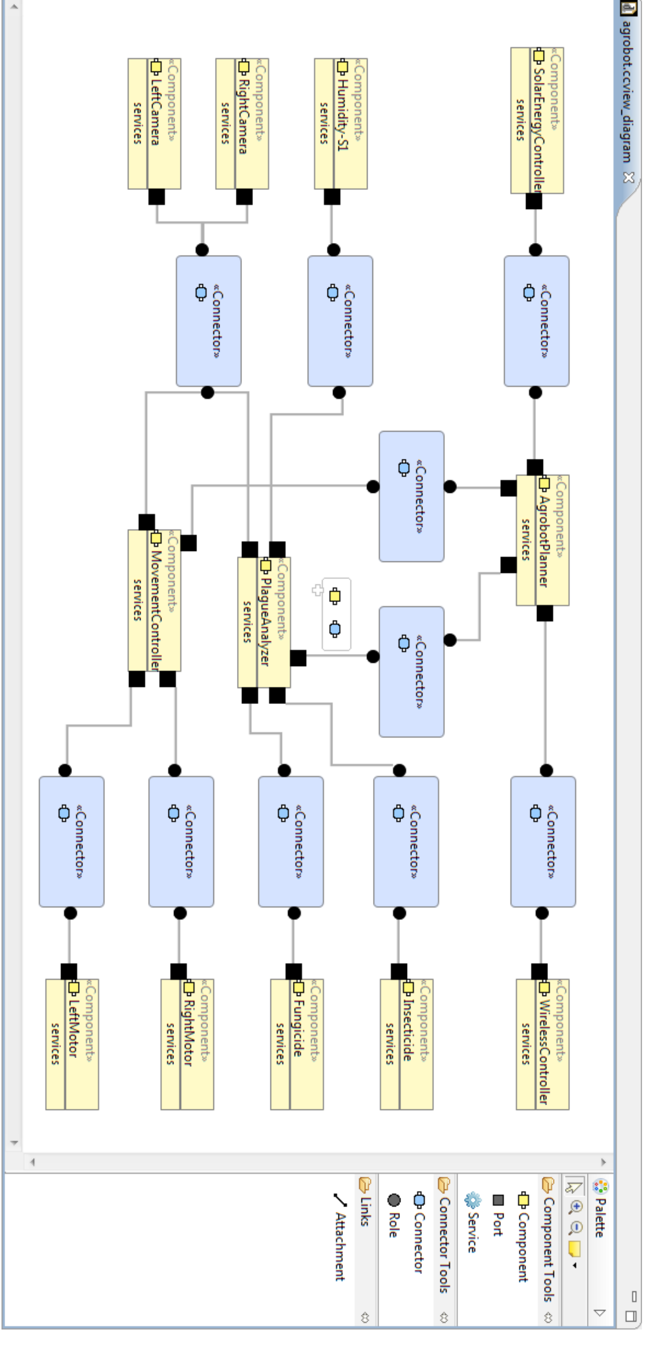


Figure 8.29: Software architecture of the Agrobot (Costa-Soria 2011)

models. As in previous metamodels, it has been built with the help of GMF and provides the classical notation for component models (i. e., components, connectors, roles, ports and attachments).

Fig. 8.29 shows an example architectural model. This example, extracted from (Costa-Soria 2011), shows the architectural model for the *Agrobot*. The *Agrobot* is an autonomous agricultural robot for plague control. It is in charge of looking for pests or disease attacks in a small field. If a threat is detected, the robot sprays insecticide or fungicide to fight the disease.

8.3.5 *PRISMA metamodel support*

PRISMA (Pérez Benedí 2006) is an architectural model based on aspects and components. It provides a DSL for components definition which allows to describe software architectures at a high level of abstraction. PRISMA integrates two approaches for systems development: the Component-Based Software Development (CBSD) (Szyperski 2002) and Aspect-Oriented Software Development (AOSD) (Filman et al. 2005).

PRISMA is an architectural metamodel developed within the ISSI research group. The metamodel implemented in MULTIPLE *is based on an* Ecore *implementation developed by Costa-Soria (2011).*

PRISMA architectural models are defined in terms of the metamodel shown on Fig. 8.30. Next, the elements that make up the PRISMA metamodel are explained.

PRISMA has three types of architectural elements: *components*, *connectors* and *systems*.

COMPONENT A component is an element that captures the system functionality. It consists of a set of aspects (functional, distribution, etc.), and one or more input and output ports, whose type is specified by an interface. Components interact with other architectural elements by means of their ports.

CONNECTOR A connector is an element of the system that acts as a coordinator between various elements. It consists of a set of aspects and a set of input and output roles that have as an interface type. Connectors interconnect and sync components, other connectors or systems by means of their roles.

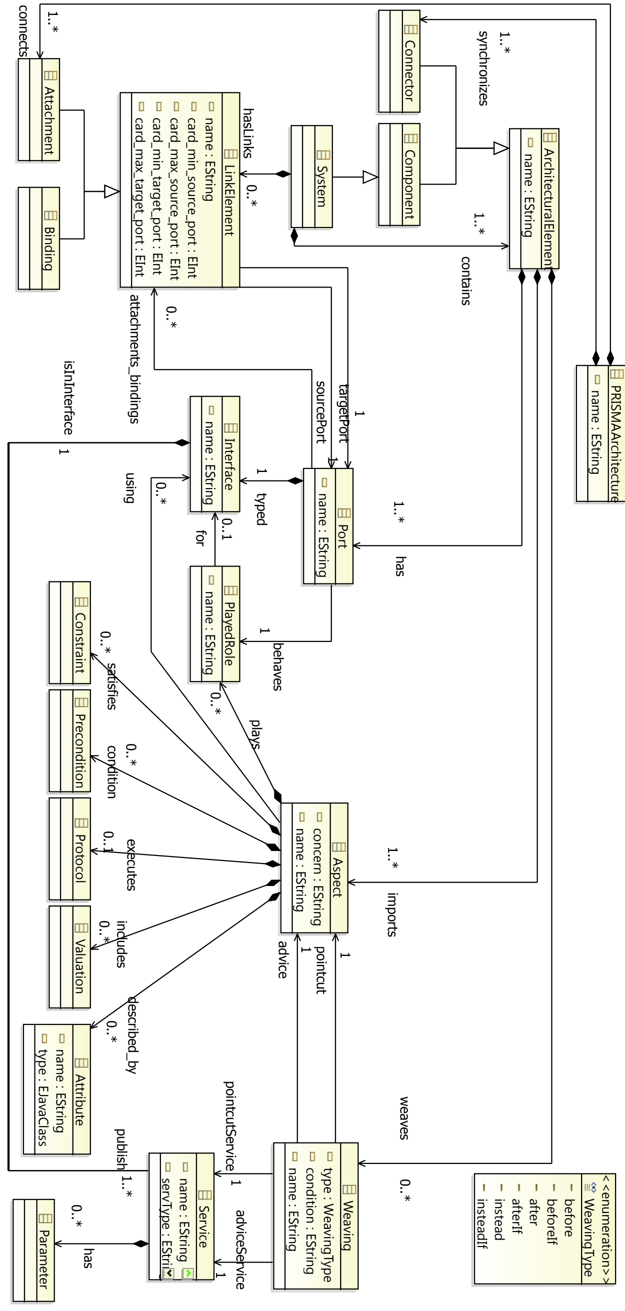


Figure 8.30: PRISMA metamodel

SYSTEM A system is a component that encapsulates a set of connectors, components and other systems properly connected. In addition, it is characterized by the definition of links (*LinkElements*) between the system ports and the ports of the components that it encapsulates.

To interconnect the architectural elements of PRISMA it provides the following elements:

LINKELEMENT establishes a connection between two architectural elements, specifically between the port of a component and the port of a connector. It can be of two different types: *Attachment* and *Binding*.

Other elements in the metamodel used to specify the architectural elements are:

PORT represents the interaction points among architectural elements.

INTERFACE provides a set of services. There are different types of interfaces, one for each type of aspect. It describes the signature of the services that can be invoked through it.

PLAYEDROLE defines the behavior of a port with a given role and the behavior of a particular interface. Moreover, it establishes how and when the services of an interface may be required or provided.

ASPECT defines the structure and behavior of a component. An aspect can be seen as the union of a set of interfaces of the type of that aspect, plus the specification of the semantics of its structure and behavior defined by: attributes, services, preconditions, valuations, constraints, roles, protocols and roles. There are different types of aspects: functional, distribution, coordination, etc.

ATTRIBUTES store the information required by aspects. Attributes can be derived or not. The value of derived attributes is calculated using a derivation rule, and this value is not stored explicitly. Non-derived attributes can be constant or variables. Constant attributes store a value which does not change, i. e., its value can not be modified at runtime. Variable attributes store a value which can be modified at runtime.

SERVICE is a process that executes a set of actions to produce a result.

PROTOCOL provides the services of an aspect that can be executed. It defines a process that coordinates the services of an aspect.

VALUATION defines the change of the state of an aspect when one of its services is executed.

PRECONDITION defines conditions to execute an action.

CONSTRAINT are conditions that must be met through whole execution of the process of an aspect.

8.3.5.1 *Internal structure*

The plugins that implement the PRISMA metamodel is built using EMF as the previously presented metamodel. Next, the plugins and summarized.

ES.UPV.DSIC.ISSI.PRISMA This plugin implements the metamodel to describe PRISMA models. The core implementation of this plugin is automatically generated by EMF, and requires the usual plugins (`org.eclipse.core.runtime` and `org.eclipse.emf.ecore`).

This plugin contains three packages:

es.upv.dsic.issi.prisma — The interfaces package.

es.upv.dsic.issi.prisma.impl — The implementation package.

es.upv.dsic.issi.prisma.util — The utility package.

ES.UPV.DSIC.ISSI.PRISMA.EDIT This plugin provides the providers to customize how the model elements are shown. The `es.upv.dsic.issi.prisma.edit` plugin has the following dependencies:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.emf.edit — The edit provider basic runtime.

es.upv.dsic.issi.prisma — The PRISMA metamodel.

The contents of this plugin are packaged in a single package (`es.upv.dsic.issi.prisma.provider`), which is automatically generated.

ES.UPV.DSIC.ISSI.PRISMA.EDITOR This plugin implements the tree editor and wizards to build PRISMA models. The editor is the default editor for **.prisma* files. The plugin has the following requirements:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.core.resources — The API to access workspace resources.

org.eclipse.emf.ecore.xmi — The plugin to support XMI.

org.eclipse.emf.edit.ui — The part of the *edit framework* which contributes to the Eclipse UI.

org.eclipse.ui.ide — The UI of the Eclipse IDE.

es.upv.dsic.issi.prisma.edit — The different providers to represent PRISMA models.

The contents of this plugin are created by EMF, and it only contains the `es.upv.dsic.issi.prisma.presentation` package.

8.3.5.2 *User Interface*

The PRISMA metamodel contributes the standard menus, wizards and editors to the Eclipse interface that EMF provides. These contributions are, mainly, the *New prisma model wizard* and the *Basic tree editor*. Both types of contributions were presented previously.

8.4 TRANSFORMATIONS SUBSYSTEM

The transformations subsystem is based on the *medini QVT* library (ikv++ 2011). As discussed in section 3.5, *medini QVT* is open source, but this license only applies to the transformations engine, and not to the additional tools it provides, such as the textual editor, debugger, etc. Therefore, it is convenient for our implementation to define a new plugin which contains the functionality of the transformations engine. The engine included in MULTIPLE has been built directly from the sources available in the public repositories. The plugin which encapsulates this functionality is `es.upv.dsic.issi.qvt.engine`.

The transformations subsystem implemented in MULTIPLE *is based in the open source transformations engine* medini QVT. *We have built different interfaces, which invoke the core engine to execute model transformations.*

To execute model transformations, the `es.upv.dsic.issi.qvt.launcher` and the `es.upv.dsic.issi.qvt.launcher.ui` plugins have been developed. Specifically, the first one encapsulates the logic to execute model transformations, while the second one implements the UI. This way, they import the `es.upv.dsic.issi.qvt.engine` plugin, and invoke the `QvtProcessorImpl.evaluateQVT(...)` method.

The launcher plugin depends on two EMF models. The first one corresponds to the QVT transformation invocation model (see sect. 8.4.2). This model allows specifying the information needed to execute a model transformation at a high level of abstraction. The second model (described in sect. 8.4.3) corresponds to the MULTIPLE traceability metamodel. This metamodel is a generic metamodel to describe a set of named links among elements of different EMF models. Moreover, the traceability metamodel provides a specific editor which allows to define and navigate these links. This way, every time a QVT transformation is executed, a traceability model is

be created describing the mappings among the source and the target models.

8.4.1 *QVT engine*

The `es.upv.dsic.issi.qvt.engine` contains all the needed libraries to execute model transformations defined in QVT–Relations. The plugin has been obtained from the sources of the *medini QVT* engine directly (this includes the OSLO sources too), since we have made slight modifications on it.

This plugin depends both on Eclipse plugins and regular Java libraries. With respect to the Eclipse plugins, dependencies are:

org.eclipse.core.runtime — The Eclipse runtime.

org.eclipse.emf.common — Common functionality to support the EMF infrastructure on Eclipse.

org.eclipse.emf.ecore — The *Ecore* metamodel.

org.eclipse.emf.ecore.xmi — XMI persistence for EMF models.

org.eclipse.emf.edit — EMF edit support subsystem.

org.eclipse.emf.transaction — Support for transactions on EMF models.

org.eclipse.emf.validation — Built-in validation capabilities for EMF-based models.

Regarding the common libraries, dependencies are:

Apache Commons Collections (Apache 2011) (commons-collections-3.2.jar).

CUP Parser Generator for Java (Hudson 2011) (CUPRuntime.jar).

Kent Modeling Framework (KMF 2011) (KMF_Util.jar, KMF_XMI.jar, KMFpatterns.jar, Util-1.2.jar)

8.4.2 *QVT transformation invocation model support*

In order to easily deal with the information needed to invoke a model transformation an *Ecore* model has been defined. This allows us to easily generate the Java code to define and query this information programmatically. Moreover, we automatically obtain the persistence mechanisms to serialize it and deserialize it.

Thus, using the generated code, different plugins can easily exchange the information of an invocation. In MULTIPLE is used for:

- First, the plugin that implements the UI to configure a new transformation (`es.upv.dsic.issi.qvt.launcher.ui`) uses the MVC pattern. Using the generated code we get the model implementation for free.

- Second, according to the Eclipse API, the class that executes an external process (`org.eclipse.debug.core.model.ILaunchConfigurationDelegate`) should receive the launch configuration data in plain text. Using the XMI persistence mechanism, the textual representation of an invocation is obtained automatically.

8.4.2.1 *Internal structure*

We have generated the model and the edit support plugins for the QVT transformation invocation model. In this case, the editor plugin is not generated as it is unnecessary (the model is only for internal use).

ES.UPV.DSIC.ISSI.QVT.LAUNCHER.MODEL The `es.upv.dsic.issi.qvt.launcher.model` plugin contains the model and the generated code for the description of QVT transformations invocations. It is shown in Fig. 8.31.

This figure shows that the model has only two classes *QvtTransformationInvocation* and *Domain*. The first one corresponds to the

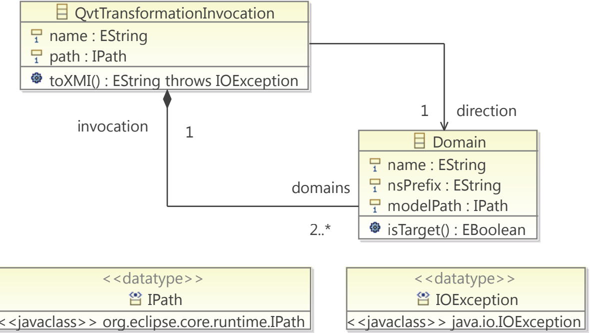


Figure 8.31: QVT transformation invocation model

transformation itself and, as can be seen, a transformation has several domains involved.

The *QvtTransformationInvocation* class also stores all the relevant information about the QVT transformation itself (i. e., the name of the transformation and the path of the file with the textual description). The *Domain* class contains the information for each domain. Specifically, the domain name, the *nsPrefix* of the metamodel that the domain model conforms to, and the path of the file containing the instance model that will match to this domain. Finally, the *direction* role which links the *QvtTransformationInvocation* class with the *Domain* class indicates in which direction the transformation is executed (i. e., which is the target domain).

The plugin has the following dependencies:

org.eclipse.core.runtime — The Eclipse runtime. It is where the *IPath* interface is defined.

org.eclipse.core.resources — The resources API plugin. It is required to access the files in the workspace.

org.eclipse.emf.ecore — The *Ecore* metamodel.

es.upv.dsic.issi.qvt.engine — The QVT engine. It provides the mechanisms to parse and analyse a textual QVT transformation.

When a transformation is parsed an Abstract Syntax Tree (AST) is built. This tree can be queried to get the necessary information to build a *QvtTransformationInvocation* with its corresponding domains.

The generated code of this plugin has been modified by hand to extend its basic functionality. All the details about the modifications can be found in (Gómez 2008). Next, a short description of the packages of this plugin is presented.

es.upv.dsic.issi.qvt.launcher.model.qvtinvocation — This is the interfaces package. It contains almost entirely the default generated code as it does not contain executable code (except for some minor changes).

es.upv.dsic.issi.qvt.launcher.model.qvtinvocation.impl — This package implements most of the functionality this plugin provides. The following modifications to the default implementation have been made:

1. The *isTarget(...)* method has been implemented in the *DomainImpl* class (it is empty by default).
2. New factories have been added to the *QvtinvocationFactoryImpl* class (an empty instance, without domains, is created by default). This way, a new instance of *QvtTransformationInvocation* can be obtained for a textual QVT transformation. This instance will be partially populated with the transformation name and path. The corresponding domains will be also contained in the *QvtTransformationInvocation* instance. The value of the *modelPath* attribute will remain undefined, as it is unknown.
3. New methods have been defined to serialize and deserialize user-defined datatypes (i. e., IPath): *convertIPathToString(...)* and *createIPathFromString(...)*

4. The implementation of the *QvtTransformationInvocation.toXMI()* method has been provided. It returns an string with the XMI serialization of the invocation.

es.upv.dsic.issi.qvt.launcher.model.qvtinvocation.util — This package contains the default implementation of the utility classes.

ES.UPV.DSIC.ISSI.QVT.LAUNCHER.MODEL.EDIT This plugin provides the basic functionality to represent the model elements in graphical UIs. Since it contains the default implementation it will not be described in further detail.

8.4.3 *Traceability metamodel support*

In *medini QVT*, treatment of traceability is done according to the recommendations of the QVT standard; i. e., it generates a *trace class* for each rule in the transformation. *Trace classes* have a property for each one of the domains of the rule. Thus, when a transformation is executed a set of *trace instances* are created, and these *trace instances* are equivalent to the relation's population of tuples.

medini QVT *provides limited support for traceability and consequently does not provide an explicit traceability metamodel. In* MULTIPLE *we have decided to adapt previous works done within the* ISSI *research group to provide the necessary tools and editors.*

However, *medini QVT* does not provide any tool for inspecting traceability models. Furthermore, the variability introduced by the existence of a traceability metamodel, which is dependent of the metamodels of the domains involved, makes difficult the creation of a generic editor for viewing and navigating such models. Therefore, we have decided to adapt the traceability metamodel used in MOMENT (Boronat et al. 2005a) and its related tools and editors, which are explained in detail in (Gómez 2005), to deal with the traceability models generated by *medini QVT*.

Next, the three plugins which provide the traceability capabilities are explained. It must be borne in mind that the plugins presented here are a reduced and improved version of the proposal made in (Gómez 2005). This way, the technical details of the plugins will not be explained in this document.

8.4.3.1 *Internal structure*

ES.UPV.DSIC.ISSI.TRACEABILITY.METAMODEL This plugin contains the generated Java code for the *Ecore* model shown in Fig. 8.32.

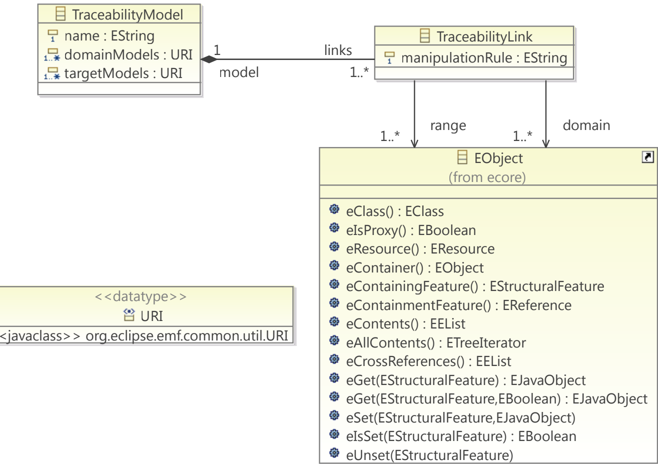


Figure 8.32: MULTIPLE traceability metamodel

As can be observed in the figure, the "root"[2] class of the model is the *TraceabilityModel* class. A traceability model has a *name* and a set of URIs (*domainModels* and *targetModels*) of the source and target domains. A URI is an indentifier used in EMF to load and reference models and metamodels (called *EResources*). A traceability model can relate n source models with n target models.

A *traceability model* contains a set of mappings or *TraceabilityLinks*. A *traceability link* has a name (*manipulationRule*, which indicates the rule that generated the link), a set of domain elements (which are instances of the classes of the domain models) and a set of range elements (which are instances of the clasess of the target models). *Domain* and *range* elements are of *EObject* type, which is

2 EMF models are typically shown as trees. This way, it is recommended to define a class which acts as the root node of this tree.

the generic type of EMF instances. The dependencies of this plugin are the following as usual:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.emf.ecore — The *Ecore* metamodel.

ES.UPV.DSIC.ISSI.TRACEABILITY.METAMODEL.EDIT This plugin provides the basic functionality to show traceability models in graphical editors. Its contents are almost the default contents.

ES.UPV.DSIC.ISSI.TRACEABILITY.METAMODEL.EDITOR This is the latest plugin developed for the traceability management in MULTIPLE. It includes the traceability editor as well as the wizard to create new traceability model automatically. The editor allows visualizing a traceability model toghether with their domain and range models simultaneously. It also allows navigating the traceability links, showing automatically and easily the relationships between the domain and range elements. Traceability models can also be edited using the editor. The only requirement of the editor to represent a traceability model that relates any set of EMF models is that the corresponding metamodels must be previously registered in EMF. The wizard allows to create new traceability models. Although traceability models are usually automatically created when a QVT transformation is executed, this feature can be useful to create new weaving models, i. e., models which define a set of associations and links among elements of different models. Weaving models can be useful for metamodel comparison, model matching, model annotation, interoperability, etc. (Del Fabro and Jouault 2005; Boronat et al. 2005a; Jossic et al. 2007).

The plugin has the following dependencies:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.core.resources — The API to access workspace resources.

org.eclipse.emf.ecore.xmi — The plugin which provides support to the XMI persistence format.

org.eclipse.emf.edit.ui — The part of the *edit framework* which contributes to the Eclipse UI.

org.eclipse.ui.ide — The UI of the Eclipse IDE, adds support for error markers, input for file editors, etc.

es.upv.dsic.issi.traceability.metamodel — The traceability metamodel plugin.

es.upv.dsic.issi.traceability.metamodel.edit — A utility plugin which implements the image and label providers for traceability models.

Furthermore, and as explained before, to be able to show the models referenced by a traceability model it is necessary that the corresponding metamodels are correctly registered in the EMF registry.

8.4.3.2 *User interface*

The traceability metamodel contributes to the Eclipse UI the standard elements. However, these elements have been customized to deal with the particularities of this kind of models. The majority of the modifications have been made in the standard tree editor. Next, this contribution is explained.

TRACEABILITY EDITOR The traceability editor has been implemented using as the starting point the code generated automatically by EMF. This code has been modified and cleaned up. Some superfluous functionaly has been removed, this way the traceability editor is a sigle-page editor and not a multi-page editor. The single-page editor has been modified, and two additional tree panels have been added at both sides of the editor. This way, the editor is able to represent three model trees simultaneously. Source models are shown in the left panel, target models are shown in the right panel, and traceability links are shown in the middle panel.

The behaviour of the editor has been also customized. When the selection changes in any of the three tree panels, the links of

the traceability model are navigated. Then, all the elements which can be reached navigating the selected elements or links are also automatically selected.

Fig. 8.33 shows an example traceability model. This model has been automatically obtained after the application of a QVT transformation. Specifically, the example transformation is the *Features-2Classes* QVT transformation explained in Sect. 7.3. The example source model is the feature model describing a product line for cars, which was explained in section 7.2.3.1 (page 89). The application of the QVT transformation generates the class diagram shown in Fig. 7.19 placed on page 109.

This way, Fig. 8.33[3] shows which elements are related in one of the applications of the *Feature2Class* rule. In this sense, can be observed that the feature *Wheel*, which is contained in the *FeatureCar*

3 The contrast of the selected elements has been artificially increased to enhance visibility in printed media.

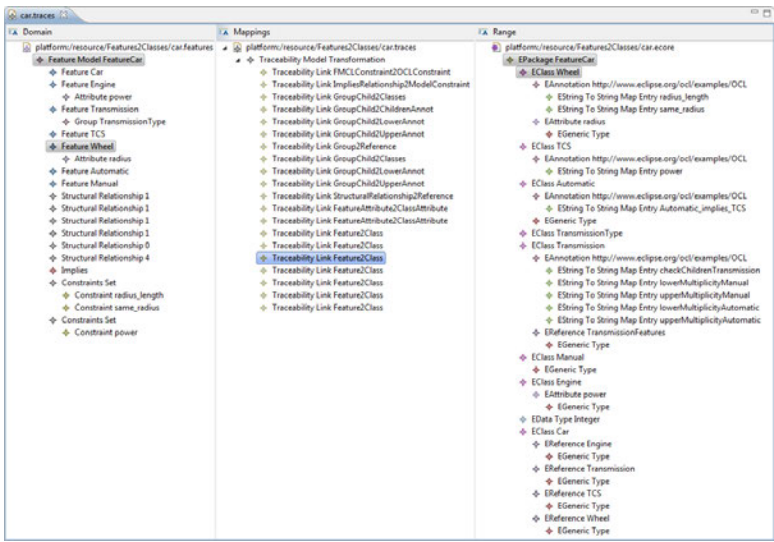


Figure 8.33: Navigating from a source element in the traceability editor

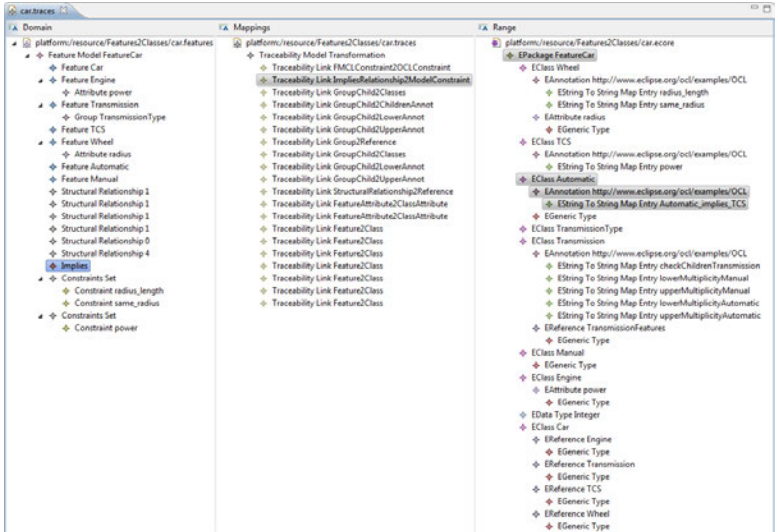


Figure 8.34: Navigating from a *traceability link* in the traceability editor

Property	Value
From	Feature Automatic
To	Feature TCS

Figure 8.35: Properties view of a domain element

feature model is transformed to the *Wheel* EClass contained in the *FeatureCar* EPackage.

Fig. 8.34 shows how the links are navigated when the user selects an element of the source model. In this case, when the *Implies* relationship is selected, the figure shows that the *Automatic_implies_TCS* annotation has been created in the *Automatic* EClass through the *ImpliesRelationship2ModelConstraint* rule.

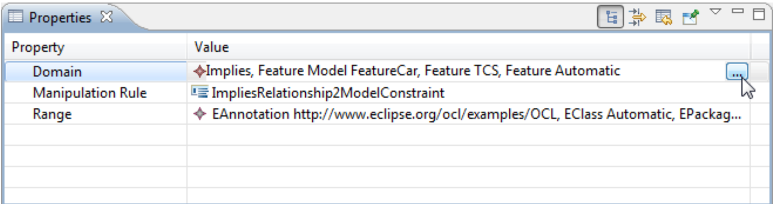


Figure 8.36: Properties view of a traceability link

Due to the genericity of the traceability metamodel, it is not always possible to show all the relevant information in the tree editor. Fig. 8.34 is an example of this. It can be observed that the label of the *Implies* element does not show which elements are linked by it. However, this information can be seen in the properties view, as it is shown in Fig. 8.35.

Fig. 8.36 shows the properties view when a traceability link is selected. The figure shows the properties of the *ImpliesRelationship2ModelConstraint* link, which relates the elements shown in

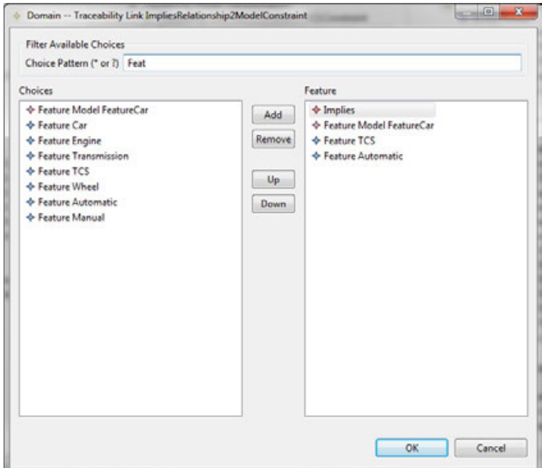


Figure 8.37: Properties view of a traceability link

Fig. 8.34, the domain property shows the referenced elements in the source models, and the range the referenced elements in the target models. The selected elements in the source and target domains can be modified filtered and modified using an additional dialog. Such dialog is opened when the [...] button is pressed. Fig. 8.37 shows the dialog for the domain elements of the *ImpliesRelationship2ModelConstraint* link.

8.4.4 *QVT Launcher*

The QVT Launcher *implements the user interface integrated into the* Eclipse *platform to invoke the embeded* medini QVT *engine.*

The previous sections have shown and described the plugins that are prerequisite to implement and execute automated model transformations. This section shows how to make use of them invoking the *medini QVT* engine and getting the result models.

8.4.4.1 *Internal structure*

The functionality to execute model transformations is divided in two different plugins. The first one, `es.upv.dsic.issi.qvt.launcher`, is able to execute a model transformation given a specific *launch configuration*, which is the generic mechanism to execute external processes within the Eclipse platform. The second one, `es.upv.dsic.issi.qvt.launcher.ui`, provides the user interface which creates the *launch configuration* and invokes the launcher plugin. Next these plugins are explained.

ES.UPV.DSIC.ISSI.QVT.LAUNCHER allows executing a model transformation without user intervention. The model transformation invocation is specified as an instance of the `es.upv.dsic.issi.qvt.launcher.model` metamodel.

This plugin makes use of the Eclipse API to execute external programs. This API specifies that a class implementing the `org.eclipse.debug.core.model.ILaunchConfigurationDelegate` interface must be declared. This class is the one in charge of executing a given *launch configuration*. This code must be implemented in the *launch(...)*

method. A launch configuration can be defined in different ways. In MULTIPLE launch configurations are defined by means of the user interface implemented in the next section, which shows up when the "*Run configurations…*" menu is selected in Eclipse.

The plugin has the following dependencies:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.debug.core — The API to deal with *launch configurations*.

org.eclipse.emf.ecore.xmi — The plugin which provides support to the XMI persistence format.

es.upv.dsic.issi.qvt.engine — The MULTIPLE plugin which provides access to the *medini QVT* engine.

es.upv.dsic.issi.qvt.launcher.model — The launcher model which allows to defines and inspect the information to execute a model transformation (QVT–Relations rules, source models, target model, etc.).

es.upv.dsic.issi.traceability.metamodel — This plugin allows to create and manipulate generic traceability models as defined in the previous section. A traceability model is always built from the information returned by the *mediniQVT* transformations engine.

The explanation of the source code of this plugin can be looked up in (Gómez 2008). Next, to avoid verbosity, only a short description of the packages of this plugin is presented.

es.upv.dsic.issi.qvt.launcher — This package only contains the activator class (*QvtLauncherPlugin*) which controls the plugin lifecycle. Moreover, this class contains some constant definitions which are used to query the details of a launch configuration.

es.upv.dsic.issi.qvt.launcher.internal — This package contains three classes: *QvtLaunchConfiguration*, *QvtTransformationProcess* and *QvtTransformationJob*.

The *QvtLaunchConfiguration* class is responsible for recovering the configuration to execute an external process, and launching it. All this is done by implementing the *launch(...)* method defined in the ILaunchConfigurationDelegate interface. The class which represents the external process is the *QvtTransformationProcess* class, which is instantiated by the *launch(...)* method.

The *QvtTransformationProcess* class inherits from the *IProcess* interface, which is the type expected by Eclipse to control the external program's life-cycle. However, the class which finally invokes the QVT engine is *QvtTransformationJob*. This class inherits from `org.eclipse.core.resources.WorkspaceJob`, which enables monitorization capabilities within the Eclipse platform.

Finally, the *QvtTransformationJob* class is in charge of invoking the *medini QVT* transformations engine. First, it navigates the *QvtTransformationInvocation* instance and gets its domains. For every domain, the corresponding model and metamodel are loaded. Models are specified by the *modelPath* attributes and metamodels are specified by the *nsPrefix* attribute. Once all the required resourced are prepared, and instance of the `de.ikv.emf.qvt.EMFQvtProcessorImpl-.EMFQvtProcessor` class is created, and its *evaluateQvt(...)* method is invoked. An example of this is shown in listing 8.1

It is noteworthy that a set of traces is returned. However, these traces cannot be represented graphically in a generic editor. Thus, these traces are next transformed to our generic metamodel. Finally, the result model and the newly created traces model are saved.

The complete code for this class can be found in Appendix F.

ES.UPV.DSIC.ISSI.QVT.LAUNCHER.UI This plugin is in charge of providing the user interface to execute model transformations. The provided interface is integrated in the dialog window to execute

Listing 8.1: *evaluateQvt(...)* method

```
1  Collection<Trace> traces = emfQvtProcessorImpl.evaluateQVT(
2    qvtScriptReader,
3        // Textual representation of the transformation
4    invocation.getName(),
5        // Name of the transformation to be executed
6    true,
7        // Set enforce execution mode
8    invocation.getDirection().getName(),
9        // Name of the target domain
10   models.toArray(),
11       // List of models to be matched with the transformation
             domains
12   new ArrayList<Trace>(),
13       // Previous set of traces. Used if the target model is
           not empty and comes from a previous execution
14   log);
15       // Log to store messages
```

external programs of the Eclipse UI, as will be shown in the next subsection.

To contribute new controls to this dialog a new class, which inherits from *org.eclipse.debug.ui.AbstractLaunchConfigurationTabGroup* must be implemented. Such class must be associated to a given launch configuration type in the manifest file of the plugin. In this case, the *QvtLaunchConfigurationTabGroup* class has been implemented. This class is related with the launch configuration type that has been implemented in the `es.upv.dsic.issi.qvt.launcher` plugin (i. e., *QvtLaunchConfiguration*).

The plugin has the following dependencies:

org.eclipse.core.runtime — The Eclipse basic runtime

org.eclipse.ui — The Eclipse basic UI library. It provides constructors to create new dialogs, etc.

org.eclipse.debug.ui — This plugins provides the utility classes to deal with the graphical aspects of launch configurations.

org.eclipse.ui.ide — This plugin provides some advanced dialog boxes.

org.eclipse.jface.text — This plugin provides a library which uses the MVC pattern and which is built on top of Standard Widget Toolkit (SWT) (Eclipse 2011d). This library is specialized in textual editors.

es.upv.dsic.issi.qvt.engine — This plugin encapsulates the *medini-QVT* engine. Allows to query and analyze QVT–Relations transformations.

es.upv.dsic.issi.qvt.launcher — This plugin provides the environment to automatically execute model transformations configurations.

es.upv.dsic.issi.qvt.launcher.model — This plugin provides the model to specify model transformations configurations.

A short description of the packages of this plugin is presented next:

es.upv.dsic.issi.qvt.launcher.ui — This package only contains the activator class, which controls the lyfe-cycle of the plugin.

es.upv.dsic.issi.qvt.launcher.ui.internal — This package contains the following classes: *ExtensionFilter*, *ParamsCellModifier*, *ParamsContentProvider*, *ParamsLabelProvider*, *QvtLaunchConfigurationTabGroup*, *QvtLaunchShortcut*, *QvtMainTab*, *ResourcesTreeSelectionDialog*.

The *QvtLaunchConfigurationTabGroup* class implements a tab group which is associated to our launch configuration type. The tab group contains a *QvtMainTab* (which implementes the main UI contributed by this plugin), and the generic Eclipse *Common* tab. The *QvtMainTab* implements a table which is used to specify the different models that will match the QVT–Relations transformation domains. The *ParamsCellModifier*,

ParamsContentProvider and *ParamsLabelProvider* classes are some utility classes used by the table contained in the *QvtMainTab* class. Finally the *ResourcesTreeSelectionDialog* implements a file selection dialog, and the *ExtensionFilter* class implements a file filter based on file extensions.

8.4.4.2 *User Interface*

The user interface that the MULTIPLE framework provides to execute model transformations is provided by the `es.upv.dsic.issi.qvt.launcher.ui` plugin. This plugin provides an easy way to define the models that take part in a model transformation. Such interface is integrated in the dialog window to execute external programs included in the Eclipse platform. This dialog is available can be opened using the *Run* → *Run configurations...* menu. Figure 8.38 shows how to launch this dialog.

If this menu is used, the *Edit configuration* dialog shown in Fig. 8.39 is shown. To be able to define the models that take part in the model transformation a **.qvt* file must be selected first.

Alternatively, and as an a shortcut, this dialog can be launched using the contextual menu which appears when right clicking over a "*.qvt" file as shown in Fig.

When this contextual menu is used, the dialog can be opened with some values already defined, such as the QVT transformation file, the

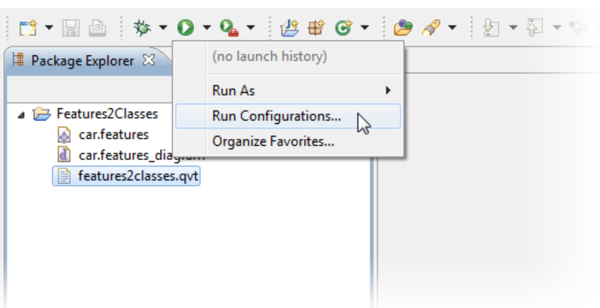


Figure 8.38: *Run* → *Run configurations...* menu

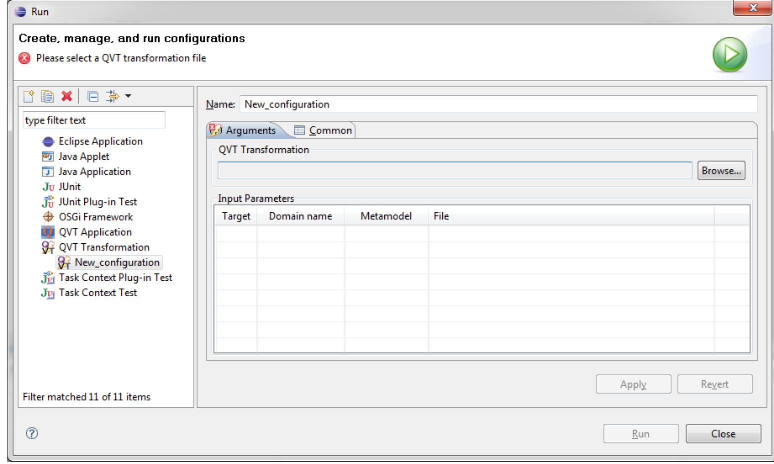


Figure 8.39: *Edit configuration* dialog

launch configuration name an the domains of the transformation. This way, the user only has to define the files that will match the transformation domains. Fig. 8.41 shows how this dialog looks like when it is opened using the contextual menu.

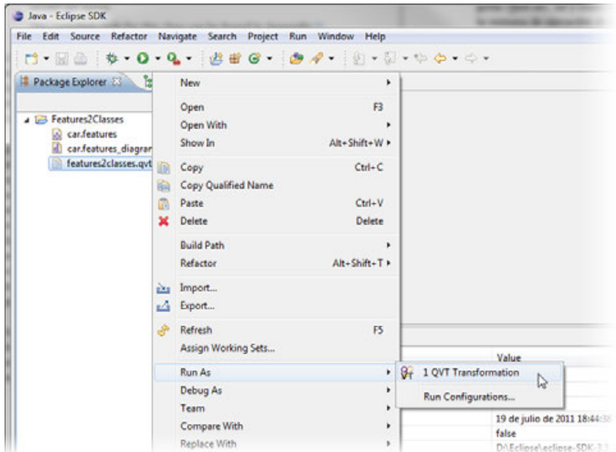


Figure 8.40: *Run as...* → *QVT Transformation* contextual menu

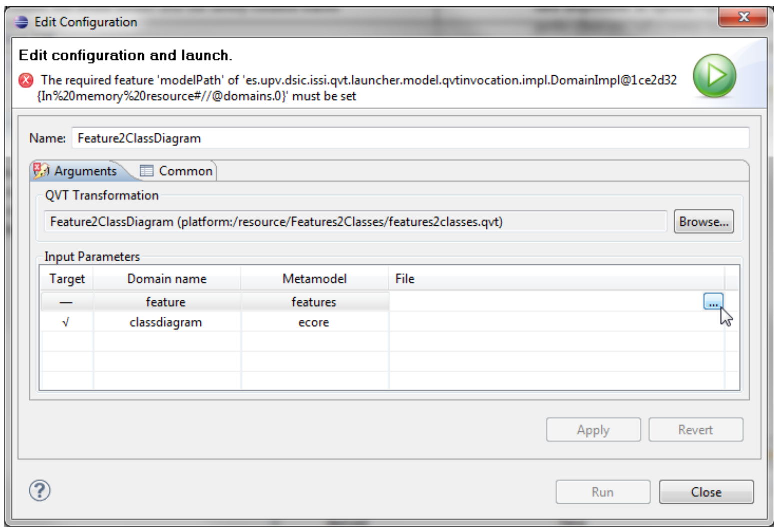


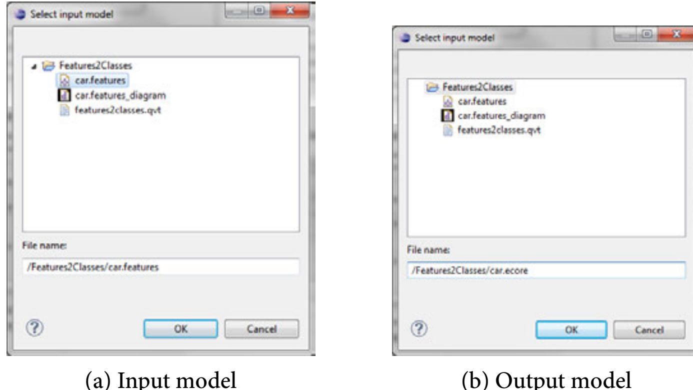
Figure 8.41: *Edit configuration and launch* dialog window

Next, the user has to define the input models that will match the different domains. For example, Fig. 8.42a shows how the feature model of the example (the car example) is selected. This dialog is launched when the user clicks on the (...) button which appears on Fig. 8.41.

When the user has defined all the input models, he/she must define which is the file name for the output file. If this file does not exists yet, its name can be entered manually as the Fig. 8.42b shows. In the example, and as the target domain conforms to the *Ecore* metamodel, the result file has the **.ecore* extension.

When all the domains have been defined, the *Run* button becomes active (Fig. 8.43). This way, it can be clicked and the model transformation begins.

Fig. 8.44 shows the progress monitor which informs the user about the progress of the model transformation. When the process finishes the progress monitor is closed and the result files are created in the Eclipse workspace.



(a) Input model (b) Output model

Figure 8.42: Setting the input and output models of a model transformation

Finally, Fig. 8.45 shows the result files (which are the selected files). As can be seen, two files are created. First, we find the *car.ecore* file, which conforms to the *Ecore* metamodel, and which has been cre-

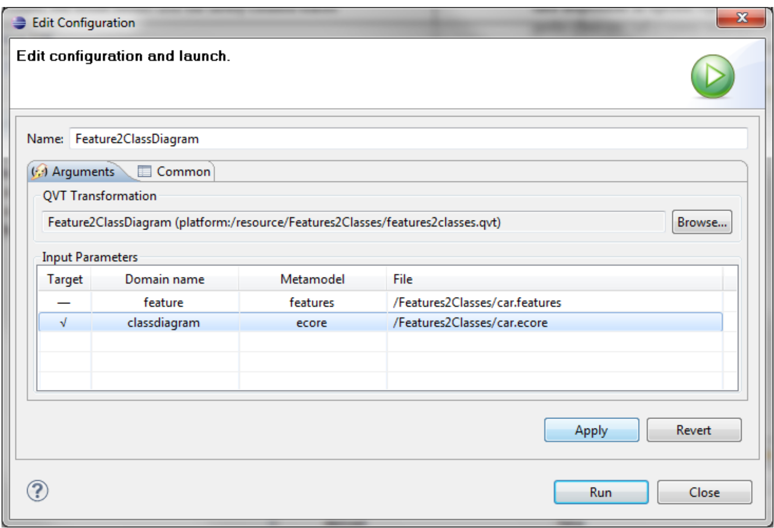


Figure 8.43: QVT transformation ready to be executed

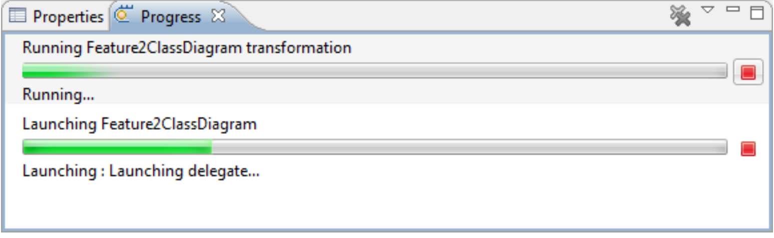


Figure 8.44: Progress monitor of a model transformation

ated following the rules that the QVT transformation define. Second, we find the *car.traces* file. This file, which is conformant to the metamodel described in section 8.4.3, makes explicit the links between the source models and the target models, which are derived from the relationships that the QVT transformation defines. The traceability model can be looked up in Figs. 8.33 and 8.34, which show the traceability editor. The contents of the *car.ecore* file are shown in the right panel of the traceability editor of both figures.

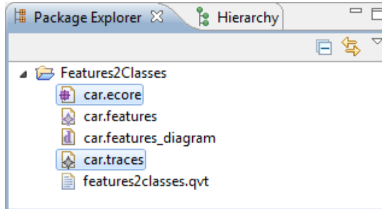


Figure 8.45: Result files of the example model transformation

8.4.5 *QVT Command-line interface*

The MULTIPLE Framework also provides support to execute model transformations outside the Eclipse platform. For this, an standalone Java program is provided. This program can execute a model transformation without user interaction if it is invoked using the proper parameters.

As it can be executed in a non-interactive way, it can be used by any other 3rd party program to integrate model transformations, apart of the technology used to develop it. For example, the *QVT Command-line Interface* can be used by a program developed using the *Microsoft .NET Framework*. An example of this case is shown in Section 13.

Next, a short overview of this program is given.

8.4.5.1 *Internal structure*

The QVT Command-line interface *is an standalone application which can execute model transformations in batch mode.*

The QVT Command-line interface is a simple program which mostly encapsulates the functionality provided by the `es.upv.dsic.issi.qvt.launcher` plugin. Although this program can be executed independently of the Eclipse workbench, some core plugins of the Eclipse platform are required. To this set of plugins some MULTIPLE dependencies must be added. All the required plugins are:

qvtemf.jar — The QVT engine packed as a single Java Archive (JAR) file.

org.eclipse.equinox.common — The Equinox common runtime, i. e., the Eclipse implementation of the OSGi framework.

org.eclipse.emf.common — Common functionality of the EMF.

org.eclipse.emf.ecore — The *Ecore* metamodel.

org.eclipse.emf.ecore.xmi — XMI support for *Ecore* artifacts.

org.eclipse.emf.edit — The EMF edit support plugin.

org.eclipse.emf.validation — The validation subsystem for *Ecore* models and instances.

org.eclipse.emf.transaction — The plugin which provides support for transactions when dealing with EMF artifacts.

es.upv.dsic.issi.traceability.metamodel — The MULTIPLE traceability metamodel.

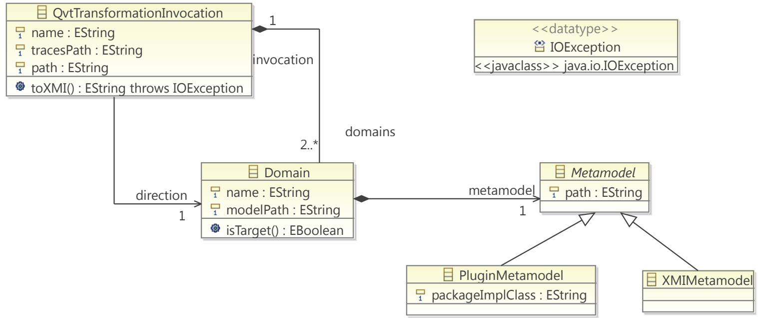


Figure 8.46: CLI invocation model

es.upv.dsic.issi.qvt.launcher.model.cli — (A customized version of) the MULTIPLE QVT *launcher model*. Fig. 8.46 shows what this modified version looks like. As it can be observed, the main difference resides in the declaration of the metamodels that the domains conform to. In the CLI version, the metamodels must be declared using a path in the file system instead of by means of their *nsPrefix*. This is because they are not loaded by default in the EMF package registry, and this task must be done explicitly.

A metamodel can be declared using two different ways. On the one hand, an XMI representation of an *Ecore* model can be used. On the second hand, an EMF model plugin can be used. This second choice can be useful to import metamodels which include executable code. In this case, the Fully Qualified Name (FQN) of the *Package Implementation* class must be provided.

The internal code of the CLI engine is stored in a single package, `es.upv.dsic.issi.qvt.cli`. This package contains two classes: *QvtTransformer* and *UnsupportedOptionException*. The latter is thrown when the user does not declare the arguments of the program properly. The former class contains the main method which parses the arguments of the program, loads the configuration, and

executes the *evaluateQvt(...)* method as explained in the previous section.

All the required plugins and libraries, within the classes of the `es.upv.dsic.issi.qvt.cli` package are built all together in what is called a *Fat JAR*. A *Fat JAR* is a single and self-contained JAR which can be easily distributed and executed without worrying about external dependencies.

8.4.5.2 *User Interface*

The program does provide a fully textual user interface. All the required information to execute a model transformation must be defined before the execution of the program, as it provides a non-interactive interface. This way, QVT transformations can take part in batch processes.

When the program is not properly configured provides the following usage information:

Listing 8.2: Usage of the CLI of the QVT engine

```
1 c:\tests> java -jar qvtengine.jar
2 Usage: java -jar qvtengine.jar --config <xmi_config_filename>
      [--debug on|off] [--traces on|off]
3 c:\tests>
```

Where the arguments are:

`--config <xmi_config_filename>` — Required. Defines the path of the XMI file which contains the information about the transformation invocation. It must be an instance of the metamodel shown in Fig. 8.46.

`--debug on|off` — Optional, default value `off`. Shows debug messages outputted by the QVT transformations engine.

`--traces on|off` — Optional, default value `off`. Creates a traceability model which contains the links that have been created between the source models and the target model. The traceability model produced conforms to the traceability metamodel presented in section 8.4.3.

Listing 8.3: Example invocation file: uml2rdbmsconfig.xmi

```
1 <?xml version="1.0" encoding="ASCII"?>
2 <qvtinvocationcli:QvtTransformationInvocation xmi:version="
      2.0" xmlns:xmi="http://www.omg.org/XMI"
3   xmlns:qvtinvocationcli="http://es.upv.dsic.issi/qvt/
        invocation/cli"
4   name="uml2rdbms"
5   path="uml2rdbms.qvt"
6   direction="rdbms"
7   tracesPath="result.traces">
8 
9   <domains name="uml" modelPath="SimpleUML.xmi">
10    <metamodel xsi:type="qvtinvocationcli:XMIMetamodel" path=
          "SimpleUML.ecore"/>
11  </domains>
12 
13  <domains name="rdbms" modelPath="result.xmi">
14    <metamodel xsi:type="qvtinvocationcli:XMIMetamodel" path=
          "SimpleRDBMS.ecore"/>
15  </domains>
16 
17 </qvtinvocationcli:QvtTransformationInvocation>
```

An example transformation invocation is shown in listing 8.3. This listing declares a transformation invocation for the transformation *uml2rdbms* declared in the `uml2rdbms.qvt` file. This file contains the classical example which transforms a class diagram to a relational database schema. The transformation is executed in the direction of the *rdbms* domain, and the traceability model that the transformation may generate will be saved in the `result.traces` file.

The transformation has two domains: *uml* and *rdbms*. The *uml* domain will match with the `SimpleUML.xmi` model, which conforms to the metamodel stored in the `SimpleUML.ecore` XMI file. The *rdbms* domain will match with the `result.xmi` model, which conforms to the metamodel stored in the `SimpleRDBMS.ecore` XMI file. As the *rdbms* domain is the target domain, the `result.xmi` file may not exist.

Fig. 8.47 shows an execution of the previous example. First, the contents of the directory are shown. There we find: (i) `qvtengine.`

`jar`, the QVT engine executable; (ii) `SimpleRDBMS.ecore`, the target metamodel; (iii) `SimpleUML.ecore`, the source metamodel; (iv) `SimpleUML.xmi`, the source instance; (v) `uml2rdbms.qvt`, the model transformation between the source metamodel and the target metamodel; and (vi) `uml2rdbmsconfig.xmi`, the invocation configuration.

Next, the transformation is executed, with the `--traces` option enabled. Finally, in the new directory listing two additional files can be seen, `result.xmi`, which contains the result model, and `result.traces`, which contains the traces model.

In case of executing the model transformation using the JAR file which implements the source metamodel, the *uml* domain should be declared as described in listing 8.4.

Fig. 8.48 shows the example execution of the model transformation when the compiled version of the *SimpleUML* metamodel is used. In this case, the metamodel is contained in the `simpleuml_`

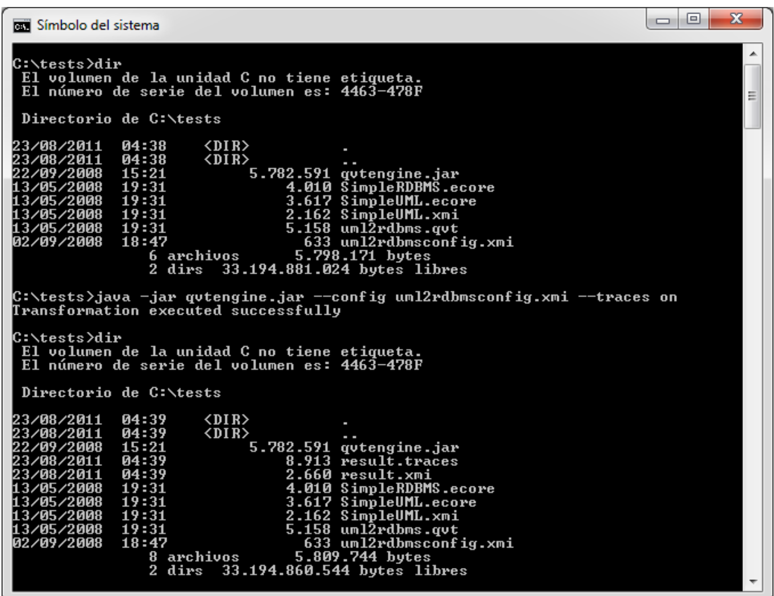


Figure 8.47: Example of a model transformation using the CLI engine

Listing 8.4: Example invocation file: uml2rdbmsconfigmod.xmi

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <qvtinvocationcli:QvtTransformationInvocation xmi:version="
       2.0" xmlns:xmi="http://www.omg.org/XMI"
3    xmlns:qvtinvocationcli="http://es.upv.dsic.issi/qvt/
         invocation/cli"
4    name="uml2rdbms"
5    path="uml2rdbms.qvt"
6    direction="rdbms"
7    tracesPath="result.traces">
8
9    <!-- Modified code begins here -->
10   <domains name="uml" modelPath="SimpleUML.xmi">
11     <metamodel xsi:type="qvtinvocationcli:PluginMetamodel"
12       path="simpleuml_1.0.0.jar"
13       packageImplClass="SimpleUML.impl.SimpleUMLPackageImpl"/
           >
14   </domains>
15   <!-- Modified code ends here -->
16
17   <domains name="rdbms" modelPath="result.xmi">
18     <metamodel xsi:type="qvtinvocationcli:XMIMetamodel" path=
           "SimpleRDBMS.ecore"/>
19   </domains>
20
21 </qvtinvocationcli:QvtTransformationInvocation>
```

`1.0.0.jar` file, and the results are the same than in the previous execution.

Although in this case there is no difference between using an XMI metamodel or a compiled metamodel, the latter is more powerful and provides more possibilities. For example, using the compiled version can be useful when metamodels have classes which contain derived attributes or methods with custom code. Moreover, it also allows including any arbitrary Java code in a model transformation, enabling the use of *black-boxes*.

```
Símbolo del sistema
C:\tests>dir
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 4463-478F

 Directorio de C:\tests

23/08/2011  05:00    <DIR>          .
23/08/2011  05:00    <DIR>          ..
22/09/2008  15:21         5.782.591 qvtengine.jar
02/09/2008  18:47             4.010 SimpleRDBMS.ecore
02/09/2008  18:47             2.162 SimpleUML.xmi
02/09/2008  18:47            35.287 simpleuml_1.0.0.jar
02/09/2008  18:47             5.158 uml2rdbms.qvt
02/09/2008  18:47               837 uml2rdbmsconfigmod.xmi
               6 archivos      5.830.045 bytes
               2 dirs  33.195.048.960 bytes libres

C:\tests>java -jar qvtengine.jar --config uml2rdbmsconfigmod.xmi --traces on
Transformation executed successfully

C:\tests>dir
 El volumen de la unidad C no tiene etiqueta.
 El número de serie del volumen es: 4463-478F

 Directorio de C:\tests

23/08/2011  05:01    <DIR>          .
23/08/2011  05:01    <DIR>          ..
22/09/2008  15:21         5.782.591 qvtengine.jar
23/08/2011  05:01             8.913 result.traces
23/08/2011  05:01             2.660 result.xmi
02/09/2008  18:47             4.010 SimpleRDBMS.ecore
02/09/2008  18:47             2.162 SimpleUML.xmi
02/09/2008  18:47            35.287 simpleuml_1.0.0.jar
02/09/2008  18:47             5.158 uml2rdbms.qvt
02/09/2008  18:47               837 uml2rdbmsconfigmod.xmi
               8 archivos      5.841.618 bytes
               2 dirs  33.195.032.576 bytes libres

C:\tests>
```

Figure 8.48: Example of a model transformation using a compiled metamodel and the CLI engine

8.5 VALIDATION SUBSYSTEM

The validation subsystem provides consistency checking and model checking capabilities to the MULTIPLE framework.

The validation subsystem allows to check it the different models and instances that play a role in a MMDSPL are correct or not. In this sense, MULTIPLE is able to check software artifacts at two levels. First, it allows to validate if models conform to their corresponding metamodels. In this conformance checking complex restrictions (expressed as OCL expressions) are included. Second, it allows to verify if cardinality-based feature models are correct or not. To guarantee that the verification process is properly done, it makes use of the FAMA framework. This framework is able to reason about feature models by representing them in different logical notations.

8.5.1 *OCL Support*

The OCL support plugin (`es.upv.dsic.issi.ocl.validator.popup`) allows to evaluate if a given instance of an *Ecore* model violates any constraint defined in it or not. This plugin makes use of the built-in OCL checker provided by EMF. The *Eclipse OCL* checker provides a full implementation of the OCL language both for *Ecore* and UML2 models. However, it does not provide a default way to define OCL expressions in such models nor a default UI to launch a checking process.

The goal of the MULTIPLE OCL Support plugin is to fill this gap. First, it defines a way to include OCL expressions within *Ecore* models, and second, it provides a simple UI to start a validation process and show the results to the user.

MULTIPLE *provides support to evaluate OCL constraints embedded in Ecore models by using model annotations. The default OCL/EMF engine is used to check the OCL expressions.*

To integrate OCL expressions within *Ecore* models, we have made use of the *EAnnotation* element. *EAnnotations* can be attached to any *EModelElement* and define a *source* attribute which identifies them. Moreover, *EAnnotations* can group any number of *detail entries*. Each one of these entries are, in the end, a pair of *key* and *value* attributes. This way, *EAnnotations* whose source attribute is `http://www.eclipse.org/ocl/examples/OCL` describe a set of OCL invariants whose context is the class containing the *EAnnotation*. Each one of the entries contained in the *EAnnotation* describe one invariant, where the *key* attribute represents the invariant name, and the *value* represents the OCL expression.

Fig. 8.49 shows how the class diagram of the example car model (shown in section 7.2.3.1) within its OCL invariants is represented in EMF. In the figure can be observed how the conventions used by the OCL Support plugin have been applied. This model represents the exact same model that the class diagram shown in Fig. 7.19 plus the OCL constraints of listing 7.4.

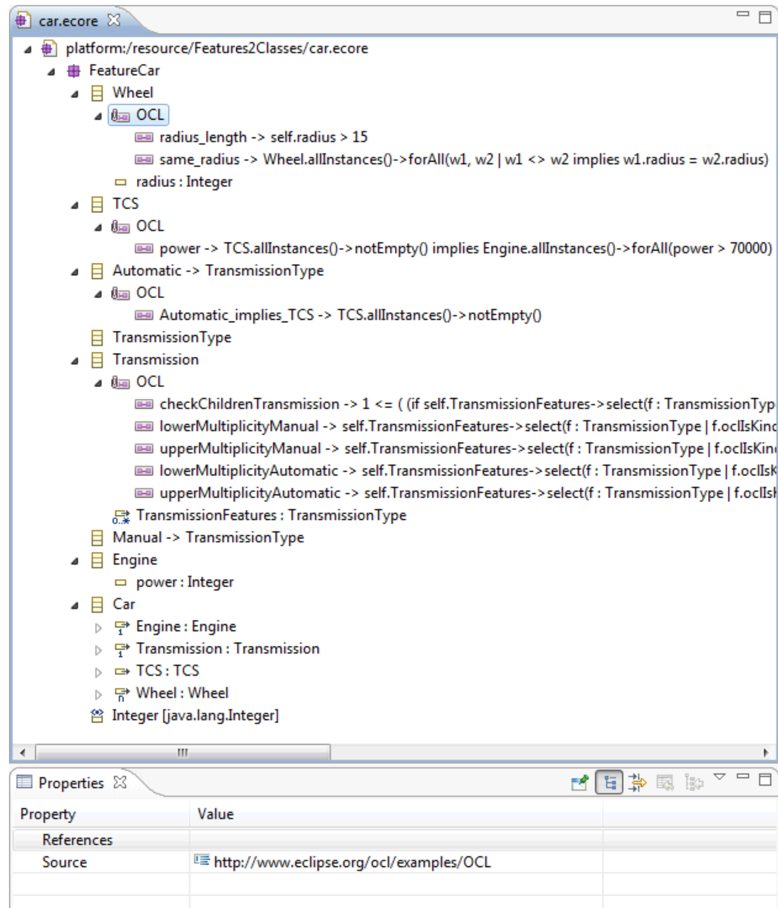


Figure 8.49: Example *car.ecore* model with OCL constraints

8.5.1.1 *Internal Structure*

The OCL support is packaged in a single plugin. Basically, this plugin provides a user-friendly interface to query *Ecore* models that are defined following the patterns expressed before, and invoke the necessary operations in the built-in OCL checker provided by EMF. This process is done using a contextual menu, and the results are shown to the user using a textual console. This plugin requires the following bundles to work:

org.eclipse.core.runtime — The Eclipse basic runtime.

org.eclipse.ui — The Eclipse UI classes.

org.eclipse.core.resources — The Eclipse API to access the resources in the workspace.

org.eclipse.emf.ecore — The *Ecore* metamodel.

org.eclipse.ocl — The generic OCL runtime.

org.eclipse.emf.ecore.xmi — The plugin which provides support to work with XMI resources.

org.eclipse.ocl.ecore — The implementation of OCL which takes into account the peculiarities of the *Ecore* metamodel.

org.eclipse.ui.ide — This plugin provides some advanced controls of the Eclipse UI.

org.eclipse.ui.console — The API to work with the integrated console view of Eclipse.

The plugin contains two packages: `es.upv.dsic.issi.ocl.validator.popup` and `es.upv.dsic.issi.ocl.validator.popup.actions`. The former only contains the activator class which controls the plugin life-cycle (*OCLValidatorPopupPlugin*). The latter contains two action classes, *ValidateAction* and *ConvertToTextAction*; and two utility classes, *OclDiagnostic* and *OclDiagnosticChain*. The *ValidateAction* launches the validation process of the selected instance in the workspace. The *ConvertToTextAction* generates a textual file containing the OCL code that contained in the *EAnnotations* of an *Ecore* model.

8.5.1.2 *User Interface and Example*

In section 7.2.3.1 an example feature model was shown. Next, section 7.3 presented a model transformation which transformed the example feature model to the class diagram shown in Fig. 7.19 and the OCL

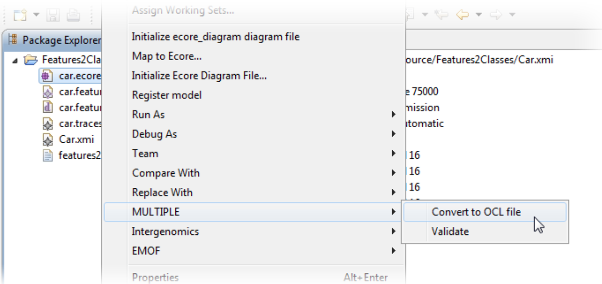


Figure 8.50: Generating an OCL textual file

constraints shown in listing 7.4. This transformation process was demonstrated in practice previously in this chapter. In section 8.3.1 and specifically in Fig. 8.12, the example feature model is represented using the MULTIPLE feture modeling editor. Section 8.4.4.2 describes how the transformations engine is configured to perform a model transformation, and specifically, the car example feature model is transformed to a class diagram. Finally, the application of the *Features2Classes* transformation over the example model produces the *Ecore* model shown in Fig. 8.49.

GENERATION OF AN OCL FILE It is possible to create automatically a full textual representation of the OCL invariants that are embedded in an *Ecore* file. Some modeling tools are able to import *Ecore* models, and those programs usually provide support to import textual files with OCL expressions. This way, the models that are created in the MULTIPLE framework can be fully imported by third party applications.

To generate the textual file with the OCL expressions, the user must right-click over an *Ecore* model, as shown in Fig. 8.50. By selecting the *MULTIPLE* → *Convert to OCL file* menu, the generation of the OCL file begins. This process, which is almost instant, generates a new file with the same name than the source *Ecore* model, but ending with the ``*.ocl'' extension.

Fig. 8.51 shows an example workspace with the automatically generated file (*car.ocl*). On the left-hand side of the figure part of the contents of the file are shown. The complete contents of the file are shown in the listing 7.4 included previously in section 7.3.

INSTANCES VALIDATION Once we have an *Ecore* model (and the needed OCL constraints embedded as *EAnnotations*), we can make use of the standard EMF tools to create instances. Fig. 8.52 shows how a dynamic instance is created for the example class model. In the case of the example model, an valid instance is equivalent to a valid configuration of the original feature model.

Fig. 8.53 shows an example instance. It shows a car configuration with automatic transmission, TCS, 4 wheels and engine. The radius of three of the wheels is 16 inches, and the radius of the fourth is 15 inches. The power of the engine is 65,000 watts. This configuration

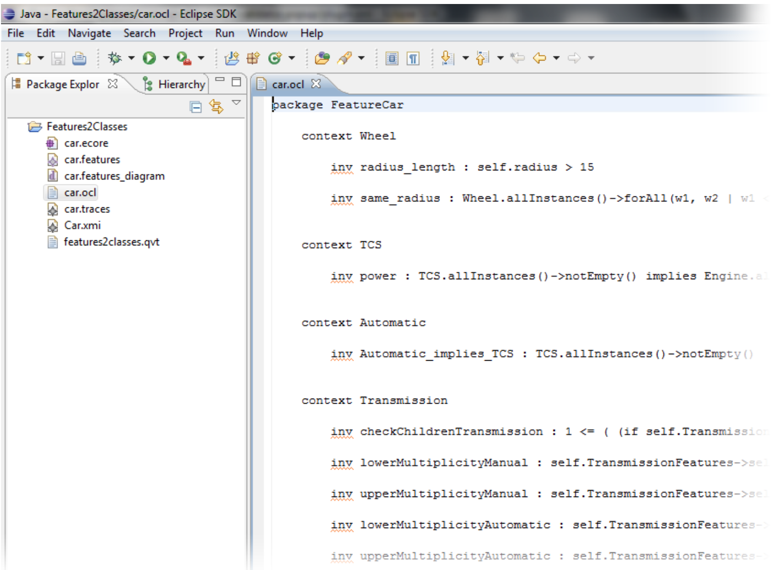


Figure 8.51: Generated OCL textual file

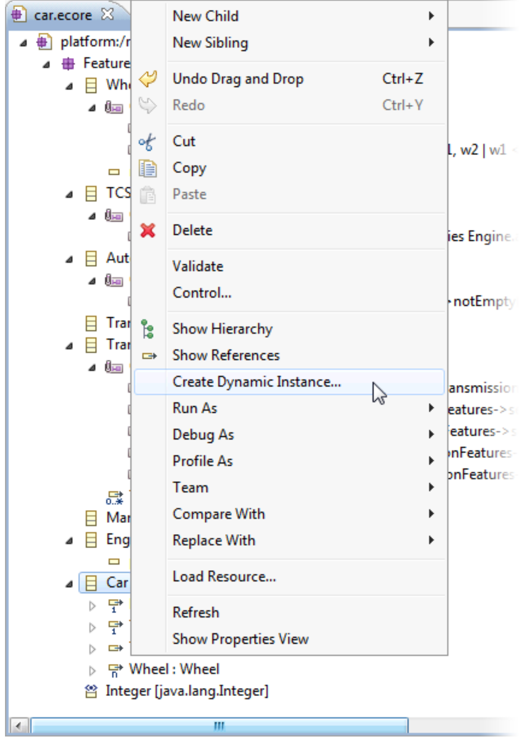


Figure 8.52: Creating an instance of an *Ecore* model dynamically

is invalid conforming to the restrictions applied to the metamodel. We can check that by using the MULTIPLE OCL checker.

To launch a validation process, we can use the contextual menu which appears when the user right-clicks over the file which contains the instance as shown in Fig. 8.54.

When the validation is perform, a dialog box appears showing the global result (i. e., if all the invariants are met) as Fig. 8.55 illustrates. When the final global result is false, it is necessary to know which invariants are not validated. To look up this information, the MULTIPLE OCL checker provides a textual console as shown in Fig. 8.56. As can be seen, in the console a message is printed for each instance that violates an invariant. Such message is made up of the following

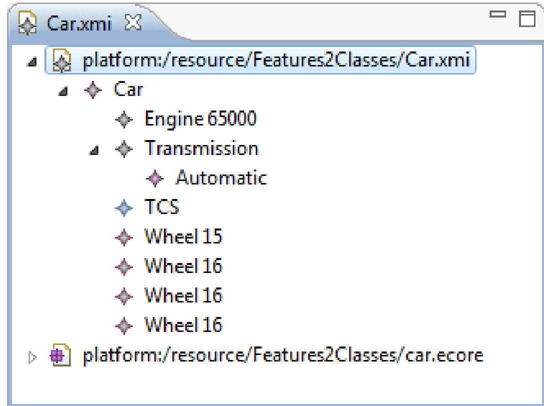


Figure 8.53: Example of a incorrectly defined instance

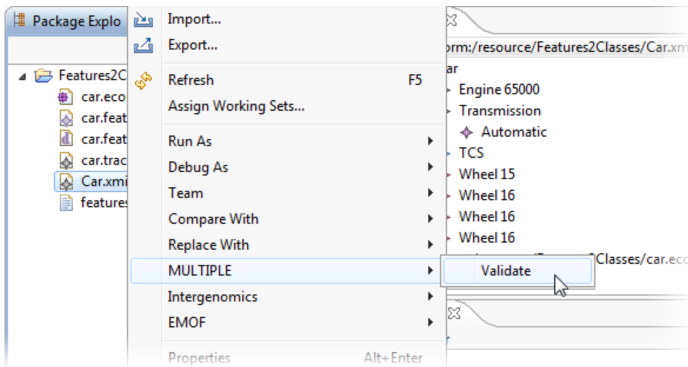


Figure 8.54: *MULTIPLE OCL checker* contextual menu

information: first, the name of the class which containts the invariant and the invariant name; second, the identifier (if available) or the URI of the object that triggered the message; and third, the result. By default, only the *false* results are shown. If the checking process is launched in debug mode, also valid checks (*true* results) are shown.

This way, the console points out the following errors: First, the wheel that sizes 15 inches does not meet the invariant expressed by the ‘‘radius_lenght’’ restriction defined in the *Wheel* class. This restriction establishes that a wheel must size more than 15 inches.

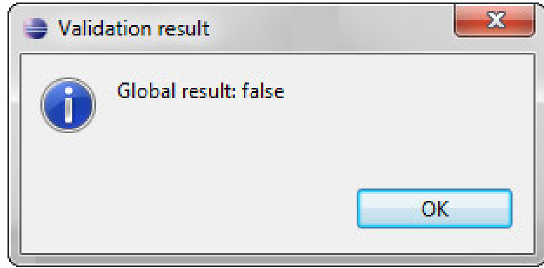


Figure 8.55: *MULTIPLE OCL checker*: unsuccessful check

```
Console
OCL Validator
[Wheel | radius_length][Car.xmi//@Wheel.0]
[false]
[Wheel | same_radius][Car.xmi//@Wheel.0]
[false]
[Wheel | same_radius][Car.xmi//@Wheel.1]
[false]
[Wheel | same_radius][Car.xmi//@Wheel.2]
[false]
[Wheel | same_radius][Car.xmi//@Wheel.3]
[false]
[TCS | power][Car.xmi//@TCS]
[false]
Global result: false
```

Figure 8.56: *MULTIPLE OCL checker*: unsuccessful check details

Second, the instance does not satisfy the "same_radius" restriction, which states that all the wheels must have the same radius. And third, it does not satisfy the "power" invariant of the TCS class, that states that if TCS is selected, the engine must be more powerful than 70,000 watts.

Once the errors presented by the console have been corrected, a new validation step can be performed. Fig. 8.57 shows how the example instance has been corrected. In this case, the power of the engine has been increased, and the wheel whose size was different has been changed. As can be seen in the figure, the validation process assures that the corrections have been properly done, and the new instance meets all the invariants that the model defines.

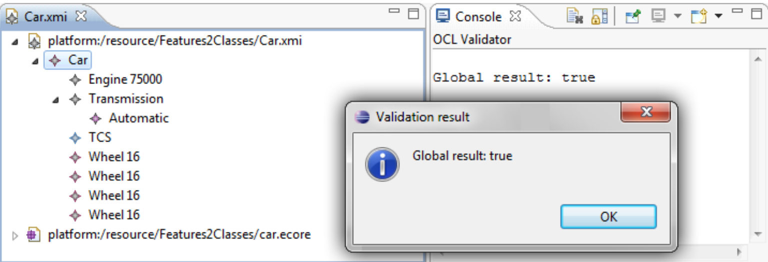


Figure 8.57: Example of a correctly defined instance

8.5.2 *OCL Support CLI*

As in the case of the transformations engine, the MULTIPLE Framework also provides a command-line version of the OCL checker that can be executed outside the Eclipse platform. For this, an standalane Java program is provided. This program can execute an instance validation without user interaction.

This program can be used by any other 3rd party program apart of the technology used to develop it. For example, the *OCL Command-line Interface* can be used by a program developed using the *Microsoft .NET Framework*. MORPHEUS is an example of such a case. This case study is presented in chapter 13.

8.5.2.1 *Internal structure*

The OCL Command-line interface (`es.upv.dsic.issi.ocl.cli`) is a small program which mostly encapsulates the functionality provided by the `es.upv.dsic.issi.ocl.validator`.popup plugin. The program is provided as a single JAR file which encapsulates some core plugins of the Eclipse platform, within the functionality provided by the MULTIPLE *OCL Support* plugin. The required plugins are:

org.eclipse.equinox.common — The Eclipse implementation of the OSGi framework.

org.eclipse.emf.common — Basic utilities of EMF (notification framework, command framework, etc.).

org.eclipse.emf.commonj.sdo — API for Service Data Objects (SDO).

org.eclipse.emf.ecore — *Ecore* metamodel implementation.

org.eclipse.emf.ecore.change — API for describing and applying model changes.

org.eclipse.emf.ecore.change.edit — Editing support for the *ecore.-change* API.

org.eclipse.emf.ecore.edit — Editing support for *Ecore* models.

org.eclipse.emf.ecore.sdo — API for supporting SDO in EMF.

org.eclipse.emf.ecore.sdo.edit — Editing support for the *ecore.sdo* API.

org.eclipse.emf.ecore.xmi — XML and XMI serialization and deserialization support.

org.eclipse.emf.edit — EMF editing support.

org.eclipse.emf.mapping.ecore2xml — API for mapping from *Ecore* constructs to the XML representation of those constructs.

org.eclipse.emf.ocl — EMF compatibility API for OCL implementation.

org.eclipse.ocl — OCL implementation.

org.eclipse.ocl.ecore — OCL implementation for *Ecore* models.

org.eclipse.ocl.uml — OCL implementation for UML2 models.

org.eclipse.xsd — API and implementation for XSD.

org.eclipse.xsd.edit — Editing support for the *org.eclipse.xsd* API.

net.sourceforge.lpg.lpgjavaruntime — Java runtime for the LALR Parser Generator (LPG) tool.

The functionality implemented by the MULTIPLE framework is grouped in a single package, `es.upv.dsic.issi.ocl.cli`, which contains the following classes: *OclEvaluator* (main class), *OclDiagnostic*, *OclDiagnosticChain* and *UnsupportedOptionException*.

8.5.2.2 *User interface*

The user interface provided by the programm is fully textual, and the program runs in a non-interactive way. Once the required arguments are properly defined, the program can be executed. This behaviour enables the reuse of this program in more complex tools.

When the program is not properly configured provides the following usage information:

Listing 8.5: Usage of the CLI of the OCL engine

```
1 C:\tests>java -jar oclevaluator.jar
2 Usage: java -jar <ocl_evaluator_jar_file> arguments
3 Required arguments:
4         --metamodel "metamodel_path"
5         --model "model_path"
6 Optional arguments:
7         --verbose [SHOW_PARTIAL_RESULT, SHOW_EXPRESSION]
```

Where the arguments are:

`--metamodel "metamodel_path"` — Required. Defines the path of the *Ecore* file containing the metamodel. It must be annoted with the OCL expressions as explained in section 8.5.1.

`--model "model_path"` — Required. Defines the path of the XMI file which conforms to the metamodel specified in the previous argument.

`--verbose [SHOW_PARTIAL_RESULT, SHOW_EXPRESSION]` — Optional. Enables the printing of extra messages about the validation process. The `SHOW_PARTIAL_RESULT` modifier enables to show the result of every single check that is performed. The `SHOW_EXPRESSION` modifier enables to show the textual OCL expression together with the partial result.

```
Símbolo del sistema

C:\tests>java -jar oclevaluator.jar --metamodel car.ecore --model car.xmi
Global result: false

C:\tests>java -jar oclevaluator.jar --metamodel car.ecore --model car.xmi --verb
ose SHOW_PARTIAL_RESULT
[Wheel | radius_length][car.xmi#//@Wheel.0]
[false]
[Wheel | radius_length][car.xmi#//@Wheel.1]
[true]
[Wheel | radius_length][car.xmi#//@Wheel.2]
[true]
[Wheel | radius_length][car.xmi#//@Wheel.3]
[true]
[Wheel | same_radius][car.xmi#//@Wheel.0]
[false]
[Wheel | same_radius][car.xmi#//@Wheel.1]
[false]
[Wheel | same_radius][car.xmi#//@Wheel.2]
[false]
[Wheel | same_radius][car.xmi#//@Wheel.3]
[false]
[TCS | power][car.xmi#//@TCS]
[false]
[Automatic | Automatic_implies_TCS][car.xmi#//@Transmission/@TransmissionFeature
s.0]
[true]
[Transmission | checkChildrenTransmission][car.xmi#//@Transmission]
[true]
[Transmission | lowerMultiplicityAutomatic][car.xmi#//@Transmission]
[true]
[Transmission | lowerMultiplicityManual][car.xmi#//@Transmission]
[true]
[Transmission | upperMultiplicityAutomatic][car.xmi#//@Transmission]
[true]
[Transmission | upperMultiplicityManual][car.xmi#//@Transmission]
[true]
Global result: false

C:\tests>
```

Figure 8.58: Example execution of the OCL CLI engine

Fig. 8.58 shows two example executions of the OCL CLI tool. The example takes as argument the example model shown in Fig. 8.53. In that figure, an incorrectly defined instance is shown. As can be seen in Fig. 8.58, the engine is first executed without any optional argument. In this case, only the global result is printed showing that the model is invalid (*Global result: false*). Next, the engine is executed again indicating that the partial results must be shown too. In this case, we can observe that the constraints that are unmet are the same as before, i. e., *radius_length*, *same_radius* and *power*.

8.5.3 *Variability Model Checking*

The variability model-checking subsystem provides validation capabilities by communicating MULTIPLE with the FAMA framework. FAMA accepts feature models in two different formats: textual and XML. MULTIPLE provides support to automatically analyse XML-based

FAMA feature models (conformant to the FAMA metamodel presented in section 8.3.2) within the Eclipse platform by using the `es.upv.dsic.issi.multiple.fama.bridges` plugin.

To support manual tasks involving FAMA the `es.upv.dsic.issi.multiple.fama.totext` utility plugin is provided. It allows translating a MULTIPLE feature model to a FAMA textual model, which can be manually edited and analysed using the standalone FAMA console.

In MULTIPLE feature model checking is performed by invoking the built-in FAMA engine.

8.5.3.1 *Internal structure*

ES.UPV.DSIC.ISSI.MULTIPLE.FAMA.BRIDGES This plugin embeds theFAMA OSGi component within MULTIPLE. It provides a set of contextual menus to analyse FAMA models with only a few mouse *clicks*.

This plugin depends on the following ones:

org.eclipse.core.runtime — The Eclipse runtime.

org.eclipse.ui — The Eclipse UI API.

org.eclipse.core.resources — The Eclipse API to access the resources in the workspace.

org.eclipse.ui.console — The console API. It allows to handle textual consoles in the Eclipse *Console view*.

es.us.isa.FaMaSDK (1.1.1) — The FAMA library. It provides the model-checking engine to represent and analyse feature models.

ES.UPV.DSIC.ISSI.MULTIPLE.FAMA.TOTEXT This plugin is in charge of representing MULTIPLE variability models using the textual representation accepted by FAMA. This format has the advantage of being compact, and easily understandable and editable (ISA 2011b, p. 12 sq.).

This plugin has the following dependencies:

org.eclipse.core.runtime — The Eclipse runtime.

org.eclipse.ui — The Eclipse UI API.

org.eclipse.core.resources — The Eclipse API to access the resources in the workspace.

org.eclipse.emf — The Eclipse Modeling Framework.

org.eclipse.emf.ecore — The *Ecore* metamodel.

org.eclipse.emf.ecore.xmi — The plugin which provides support to the XMI persistence format.

es.upv.dsic.issi.multiple.features — The MULTIPLE features metamodel.

8.5.3.2 *User interface*

The varibility model-checking capabilities are accessed by using contextual menus. Fig. 8.59 shows an example feature model as represented in the MULTIPLE graphical editor. The top feature of the model is Root, which has three children features: A, B and C. The former is an optional feature, and the last two are mandatory. A has an alternative child group, where only feature A1, A2 or A3 can be selected. Finally, and to illustrate the validation capabilites, an excludes relationship is defined between B and C. Such a relationship makes the model invalid.

Next, how this sample feature model is processed and analysed is shown.

THE FAMA CONTEXTUAL MENU To analyse a feature model using the FAMA contextual menu it is necessary to transform a MULTIPLE feature model into a FAMA feature model. This can be achieved by executing the QVT transformation shown in section 8.3.2.2. Listing 8.6 shows the generated XML file, and Fig. 8.60 shows it as it is represented by the MULTIPLE built-in graphical editor.

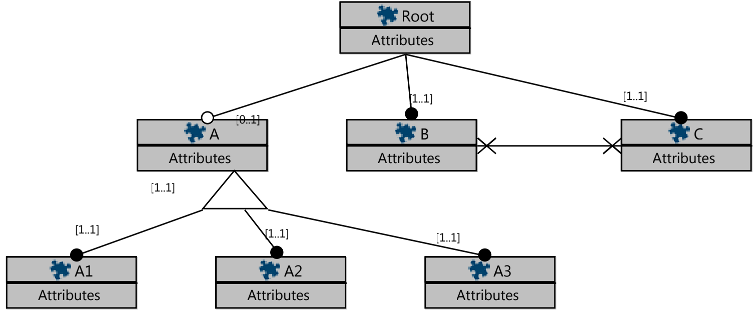


Figure 8.59: Sample void feature model

Listing 8.6: Sample void feature model represented in FAMA-XML

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <feature-model xmlns:xsi="http://www.w3.org/2001/XMLSchema-
       instance"
3  xsi:noNamespaceSchemaLocation="http://www.tdg-seville.info/
       benavides/featuremodelling/feature-model.xsd">
4    <feature name="Root">
5      <binaryRelation name="Relation_to_A">
6        <cardinality max="1" min="0"/>
7        <solitaryFeature name="A">
8          <setRelation name="Grouped_Relation">
9            <cardinality max="1" min="1"/>
10           <groupedFeature name="A1"/>
11           <groupedFeature name="A2"/>
12           <groupedFeature name="A3"/>
13         </setRelation>
14       </solitaryFeature>
15     </binaryRelation>
16     <binaryRelation name="Relation_to_B">
17       <cardinality max="1" min="1"/>
18       <solitaryFeature name="B"/>
19     </binaryRelation>
20     <binaryRelation name="Relation_to_C">
21       <cardinality max="1" min="1"/>
22       <solitaryFeature name="C"/>
23     </binaryRelation>
24   </feature>
25   <requires feature="A1" name="A2_requires_A1" requires="A2"/
         >
26   <excludes excludes="B" feature="C" name="
         Excludes_from_B_to_C"/>
27 </feature-model>
```

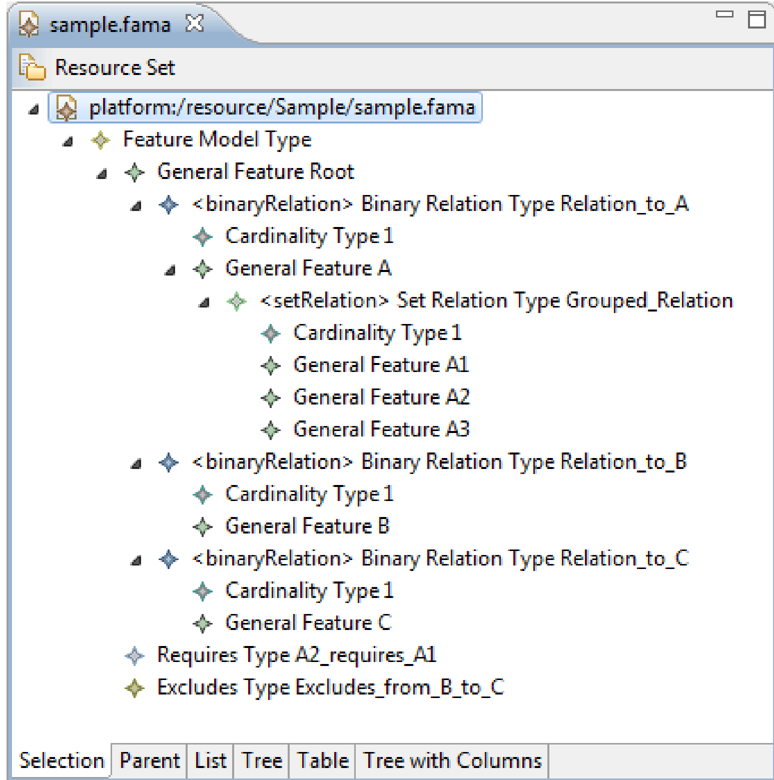


Figure 8.60: Sample void feature model represented in FAMA

Once a MULTIPLE variability model is transformed into a FAMA variability model, the FAMA contextual menu can be used to perform the different analysis. Fig. 8.61 shows how the "Detect and explain errors" analysis is launched.

The result of the analysis (Fig. 8.62) shows that the sample feature model is void, i. e., there does not exist a product that fulfills the constraints of the feature model. Specifically, it states that *Excludes_from_B_to_C*, *Relation_to_B* and *Relation_to_C* are contradictory.

MULTIPLE FEATURE MODEL TO FAMA TEXTUAL MODEL To transform a MULTIPLE feature model to a FAMA textual model a

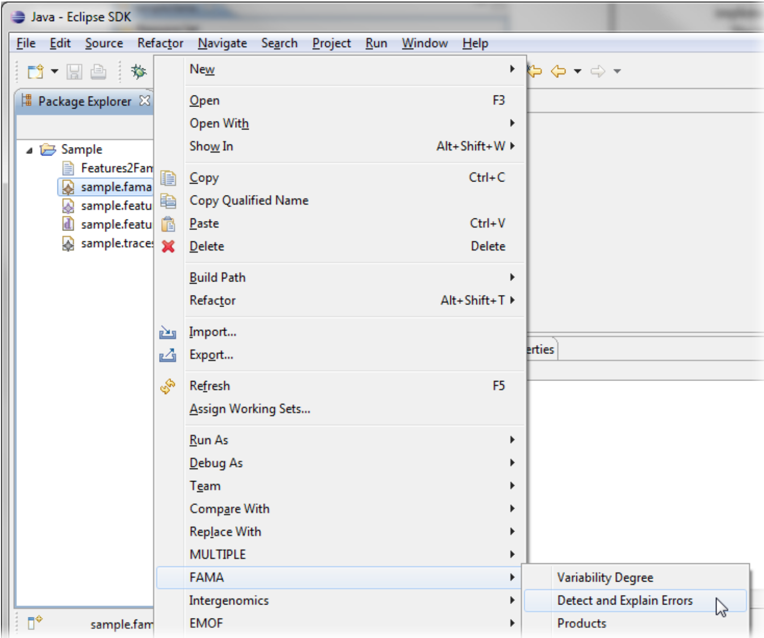


Figure 8.61: *Detect and explain errors* contextual menu

contextual menu is also used. Fig. 8.65 shows what this menu looks like. The transformation is performed almost instantly, and Fig. 8.66 shows the resulting file. As it can be observed, the textual representation is much more simple than the XML-based one.

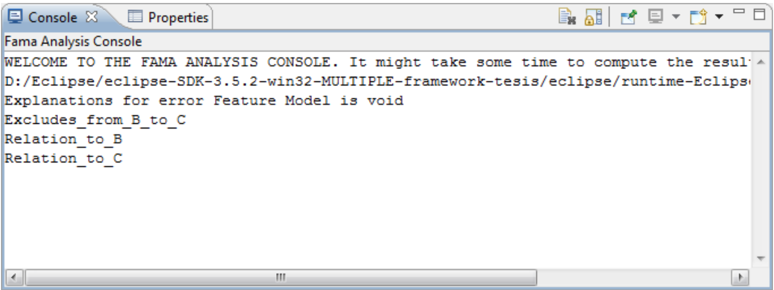


Figure 8.62: Result of *detect and explain errors*

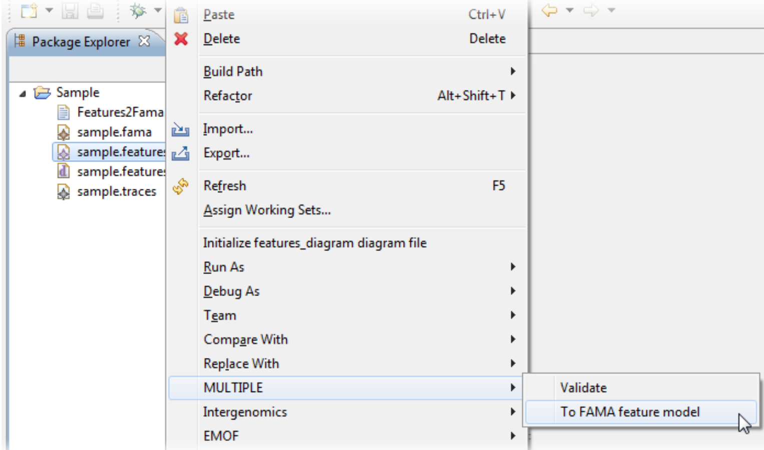


Figure 8.63: *To FAMA feature model* (as text) contextual menu

sample.fm

```
1 %Relationships
2 Root: [A] B C ;
3 A: [1,1]{A1 A2 A3 };
4
5 %Constraints
6 B EXCLUDES C;
7
```

Figure 8.64: Sample feature model in the FAMA textual representation

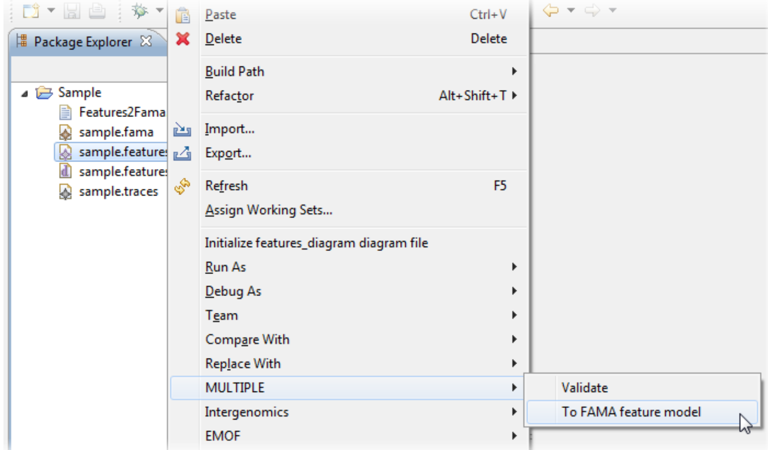


Figure 8.65: *To FAMA feature model* (as text) contextual menu

sample.fm

```
1 %Relationships
2 Root: [A] B C ;
3 A: [1,1]{A1 A2 A3 };
4
5 %Constraints
6 B EXCLUDES C;
7
```

Figure 8.66: Sample feature model in the FAMA textual representation

8.6 MULTIPLE EMF UTILS

This section describes some utility plugins which are bundled within the MULTIPLE framework. These plugins extend the basic EMF functionality and are not required to run the MULTIPLE framework. As such, they are only briefly described in a few words.

8.6.1 *EMOF Converter utility*

XMI is the canonical representation for any EMF artifact. However, the default XMI representation used by Eclipse does not validate the standardised specification provided by the MOF standard. Although, Eclipse is able to deal with standard XMI files, there is not a shortcut to easily transform EMF-XMI files to MOF-XMI files and vice versa. The `es.upv.dsic.issi.emof.converter` plugin provides a menu to switch between the two representations. These options are grouped inside an *EMOF* contextual menu with two sub-options: ``*Save as Ecore...*'' and ``*Save as EMOF...*''.

8.6.2 *Register EMF utility*

EMF maintains an internal registry of the installed metamodels. Installed metamodels can be instantiated, used, queried and referenced automatically. To install a metamodel it is necessary to perform some complex tasks, such as generating code, exporting plugins, and installing plugins. This workflow is impractical when dealing with models which are constantly changing. The `es.upv.dsic.issi.moment.registeremf` plugin allows to simmulate that a metamodel is installed without the necessity of generating code or installing plugins. Using this plugin a new metamodel can be ``installed'' in the internal registry by selecting the ``Register metamodel'' option using the contextual menu over an ``*.ecore'' file. Changes performed in the registry using this mechanism are not persistent, i. e., they are lost once Eclipse is restarted.

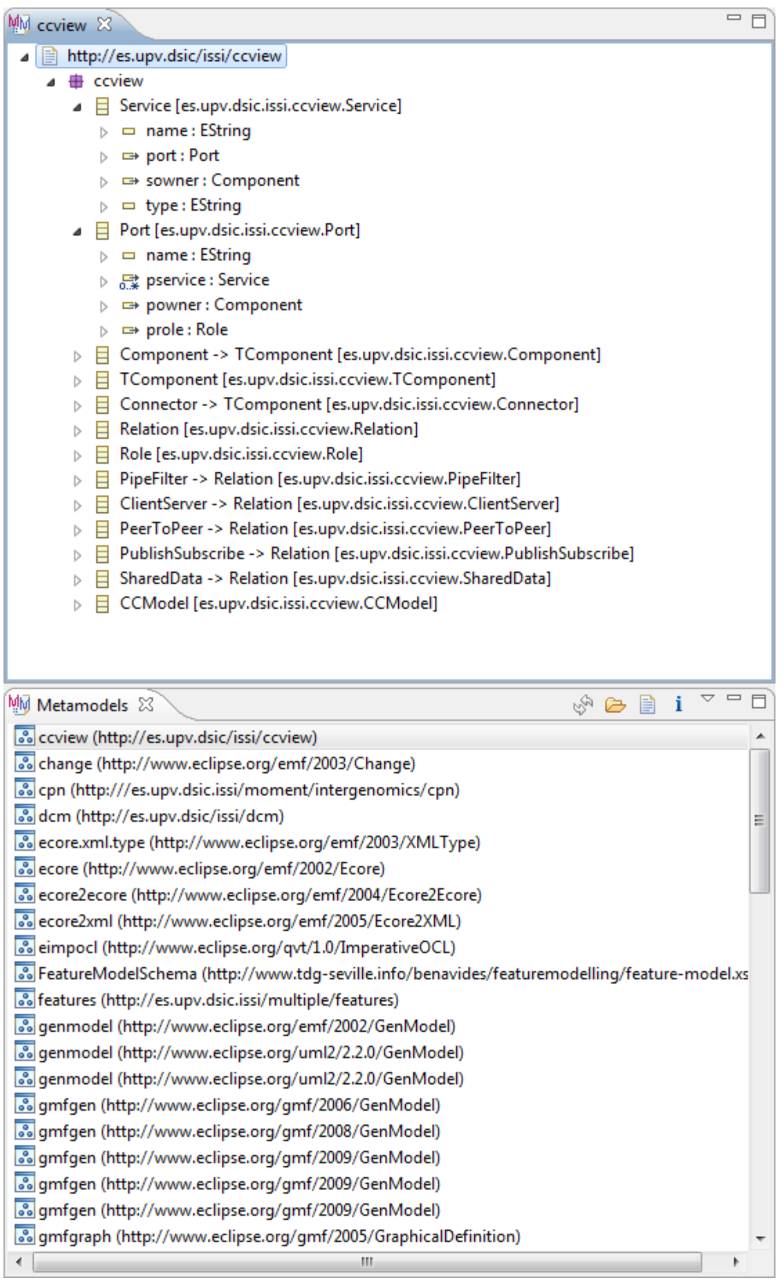


Figure 8.67: *Metamodels* view and *Metamodel tree editor*

8.6.3 *Registry viewer utility*

As explained before, EMF has an internal registry of the registered metamodels. However Eclipse does not provide a simple UI to look at this registry. The `es.upv.dsic.issi.moment.ui.registeredmodels` plugin provides a simple Eclipse *view* which lists the contents of such a registry. Using this view is also possible to easily open a registered metamodel in a tree editor to look at its contents. Fig. 8.67 shows what this view and the tree editor look like.

8.7 SUMMARY AND CONCLUSIONS

This chapter has presented the MULTIPLE framework, an Eclipse-based generic framework to describe Multi-Model Driven Software Product Lines (MMDSPLs). Using EMF and it related tools it provides a set of built-in metamodels together with the corresponding tree editors and graphical editors to specify different system views as described in chapter 5. Thanks to the variability metamodel MULTIPLE can not only be used to implement simple MDE processes, but to implement and analyse complex SPLs.

The MULTIPLE framework provides: (i) a metamodel to describe systems' variability by using rich feature models; (ii) a metamodel to describe functional views of software systems; (iii) a metamodel to define architectural descriptions of software systems; (iv) a metamodel to describe PRISMA architectural models, which allow to define executable architectural models; (v) a transformations subsystem which is able to execute QVT–Relations model transformations; (vi) a validation subsystem, which is able to perform both conformance-checking and model-checking operations; (vii) a standardized way to interchange data and metadata among tools thanks to XMI; and (viii) a extensible architecture thanks to Eclipse and its OSGi subsystem.

All this capabilities are demonstrated in the remaining of this thesis. Next chapters describe all the different case studies where the MULTIPLE framework has been used as a platform to implement

both MMDPLE or basic MDE processes. Chapter 9 focuses in multi-modeling, variability and configuration aspects. Specifically it describes how a traditional MDSPL is transformed into a MMDSPL, and how the production plan is then managed by using different QVT–Relations transformations. Chapter 10 focuses on validation and model-checking aspects. This chapter describes how the tools that the MULTIPLE framework provides have been used to analyse a large-scale feature model of an industrial SPL. Chapters in part V focus on the genericity and extensibility of the framework. This is done by demonstrating the applicability of the MULTIPLE framework in third party projects. These chapters show how the tool has been used as a suitable environment to implement different MDE processes in domains in domains as diverse as bioinformatics (chapter 11), software measurement (chapter 12) and requirements elicitation and software architectures (chapter 13).

9

MULTIPLE IN PRACTICE: MULTI-MODEL DRIVEN SOFTWARE PRODUCT LINE FOR DIAGNOSTIC EXPERT SYSTEMS DEVELOPMENT

«An expert is someone who has succeeded in making decisions and judgements simpler through knowing what to pay attention to and what to ignore»

— Edward de Bono
Maltese physician, author, inventor, and consultant, 1933–

The development of Expert Systems (ES) (Giarratano and Riley 2005) has become increasingly important in recent years creating a need to properly support such applications. Since systems are more and more relevant, the need for techniques for their development has also become more important. Additionally, ES introduce a difference regarding the decision making process: they store expert knowledge in a Knowledge Base. However, these systems are complex because their architectural elements vary.

To cope with this variability problem, SPLs emerge in an effort to control and minimize the high costs of the software development process and to reduce the time to market of these new products. As has been discussed extensively in previous sections, this approach is based on having a base design that is shared by all the product family

members. Thus, a specific product can benefit from the common parts of the software. The base design can be re-used in different products by adding different features that characterize them.

Moreover, automation and the use of open and standard mechanisms are desirable in software development to deal with the complexity of software systems. However, the automatic generation of such systems is only possible when there is a framework that supports this process. In this sense, the Model Driven Architecture (MDA) proposed by the OMG advocates the use of standards and platform independence in the software development process as a new way of producing applications.

BOM is a MDA-based framework for the development of expert systems by using SPL techniques. Throughout this chapter we will explain both the BOM–Eager and the BOM–Lazy approaches and their differences.

In this context, the Baseline Oriented Modeling (BOM) framework, a MDA approach based on SPL for applications development, is proposed in (Cabello Espinosa 2008). This work follows the MDA approach in order to automatically generate code from models, by means of transformation rules and SPL techniques, to minimize the variability impact on the cost of the software production. The PRISMA framework (Pérez Benedí 2006) is the selected target platform, and the diagnostic Expert System domain is the domain used to validate the proposal. Therefore, BOM automatically generates Diagnostic Expert Systems (as PRISMA architectural models) based on SPL by using the generative programming approach of the MDA trend. This framework based on SPL and MDA is the case study where the methods and techniques proposed by this thesis have been applied to demonstrate the benefits of the proposal.

The products of the BOM SPL have been designed as PRISMA models that capture the architecture and functionality of the rule-based ES. The process of creating a SPL uses a set of reusable resources or assets (core assets) to create a family of software products, using two OMG standards: the Reusable Asset Specification (RAS) (OMG 2005b) and the Software & Systems Process Engineering Meta-Model (SPEM) (OMG 2008c).

As has been extensively remarked throughout this work, the key element of a SPL is how to represent and manage variability an how it impacts in the rest of artifacts which which participate in the SPL. The

initial proposal of BOM, BOM–Eager, is implemented using classic MDA and SPL techniques. However, this approach presented some limitations when dealing with large-scale product lines. In order to improve the BOM framework we built BOM–Lazy. This prototype outperforms the initial proposal in some aspects by using MMDPLE techniques.

The diagnostic ES example is used to illustrate BOM. The main goal of this kind of systems is to capture the state of an entity from a series of data (observation variables) and to produce a diagnosis. The domain of expert systems for diagnosis includes systems for medical diagnosis, educational diagnosis, and emergency diagnosis, among others. In this thesis we present the medical diagnosis as the application domain, using a case study of infantile infectious diseases.

BOM has been built to achieve the following goals:

1. create new (diagnostic) systems in different domains,
2. decrease production costs by reusing software packages or assets,
3. generate code automatically to increase the productivity and quality of software and to decrease the time to market,
4. construct systems in a simpler way by using diagnosis and application domain models closer that are to the problem domain and facilitating user interaction,
5. develop platform-independent systems from the problem perspective and not from the solution perspective, which will provide generality in the development approach and applicability in different domains and platforms.

9.1 TECHNOLOGICAL SPACES

In order to deal with the complexity of the problem, this work integrates various *technological spaces* (Kurtev et al. 2002). In this sense,

a technological space is *a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities*. These are described in turn:

- The MDA as an approach to the MDE trend which is promoted by theOMG for software system development.
- The SPL approach, which, from a practical point of view is one of the most successful ones since it combines systematic development and reuse of assets; i. e., the products are different in some features but share a basic architecture.
- The PRISMA framework (Pérez Benedí 2006), which defines the architectural elements (components, connectors, and systems) through their aspects.
- The Expert Systems (Giarratano and Riley 2005), which capture the knowledge of experts and try to imitate their reasoning processes when solving problems in a specific domain.

9.2 FIELD STUDY: DIAGNOSTIC EXPERT SYSTEMS

The work presented in this thesis is based on a field study performed in the domain of Diagnostic Expert Systems performed by Cabello Espinosa (2008).

Cabello Espinosa (2008) presents a field study to learn about variability in Diagnostic Expert Systems. A subset of the Expert Systems domain has been chosen to describe our approach: the ES that are used in diagnostic tasks, the so-called Diagnostic Expert Systems (DES). The diagnosis of an entity lies on the evaluation of its state by interpreting its properties. The cited study allows us to know the DES behaviour and structure in several specific domains. The following examples were considered in the study: medical diagnosis, diagnosis of victims in disasters, television diagnosis, educational program diagnosis, and scholarship candidate diagnosis. The remainder of this chapter will refer to two paradigmatic cases: systems for medical diagnosis and systems for educational diagnosis.

In the medical diagnosis example, the entity to be diagnosed is the patient and the result of the process is the disease he/she suffers.

First, a clinical diagnosis is performed, which must be validated then by a laboratory-based diagnosis. In the end, both diagnosis are merged in a final diagnosis where the previous ones are taken into account. Thus, we can identify three basic functionalities: *get laboratory diagnosis*, *get clinical diagnosis*, and *get diagnosis*. The first one is used by the *laboratory assistant* and the last ones are used by the *doctor*. In this case, the properties of the entities considered in the process vary during the whole process, which implies the existence of several hypotheses that must be evaluated to determine the valid one using differential reasoning.

In the educations reasoning example the entity to be diagnosed is a post-graduate educational program where several quality criteria are evaluated, and the result of the diagnosis is the advance of the given program. The properties of the entities remain the same throughout the diagnostic process, therefore only one hypothesis is created applying deductive reasoning. In this case the DES only performs one task: *get program advance*, which is invoked by the user of the tool.

9.2.1 *Diagnostic Expert Systems Reference Architecture*

In SPL, there are parts that are shared among all the products, but some other parts vary from one product to another. In BOM the common parts are represented by the *reference architecture*, which captures the shared functionality. The variable part shows additional features that are specific for some products, and such parts are represented by the *base architecture*. The *reference architecture* of DES is expressed in our approach by a modular model made up of three basic modules (see Fig. 9.1):

The reference architecture represents the structure that all the members of the SPL share.

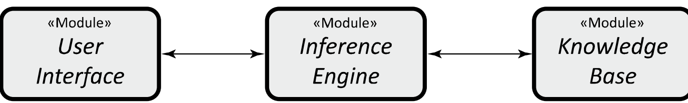


Figure 9.1: Reference Architecture of Expert Systems

- *Inference Engine* module. This module contains the inference process that solves a problem in a specific domain.
- *Knowledge Base* module. The *Knowledge Base* module contains the knowledge about the domain
- *User Interface* module. This module allows the communication between the user(s) and the system.

The reference architecture is used as the shared structure of an application that is member of the product line, but, there also exist additional features that are particular to a specific application. This implies the creation of a specific base architecture when a product of the SPL is obtained from the reference architecture. However, the base architecture that is generated from the reference architecture is not unique, because systems vary not only in their structure but also in their behaviour as explained in the following subsection.

9.2.2 *Diagnostic Expert Systems Structural Variability*

To illustrate how the architectural elements of a DES vary in their structure, we have modeled the functional requirements that the final product must satisfy using UML Case Diagrams. These diagrams show the different functionality that the user expects from the system and how the system interacts with its environment. In particular, the structure of the architectural elements vary according to the number of use cases, the number of actors and the number of use cases that are accessed by each actor.

Although diagnostic expert systems share a reference architecture, they can vary in their structure, their behaviour and in their application domain.

Fig. 9.2 shows the use case diagram for the medical diagnosis domain together with its corresponding base architecture. As Fig. 9.2a shows, the ES for medical diagnosis of our case study has two actors: *doctor* and *lab. assistant*. The first one uses the system to get clinical diagnosis and the final diagnosis; and the second one uses the system to get the laboratory diagnosis. These use cases affect the final base architecture of the system. Since we implement the

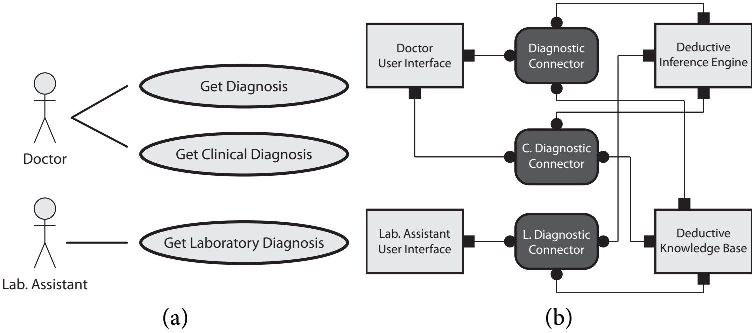


Figure 9.2: Medical diagnosis use case diagram (a) and its corresponding base architecture (b)

SPLs assets using PRISMA software architectures, i. e., PSM, the modules of the Expert System generic architecture must be mapped into the following architectural elements: the *Inference Engine* Component(s), which establishes system control and provides the general resolution strategy for taking a decision; the *Knowledge Base* Component(s), which contains the domain knowledge of the case study using application domain rules (Horn clauses) and facts (constant information); and the *User Interface* Component(s), which establish(es) the human-computer interaction. For example, Fig. 9.2b shows that a user interface module is used for each one of the actors shown in the diagram. Therefore, there is a correspondence among the modules and their respective components. However, to be consistent with the PRISMA metamodel, it is necessary to incorporate a new architectural element (connector) to establish the communication among components.

9.2.3 *Diagnostic Expert Systems Behavioral Variability*

Behavioral variability of the architectural elements of an ES depends on the type of diagnosis, and therefore the reasoning to be used. As presented in the previous section, the inference process to apply is

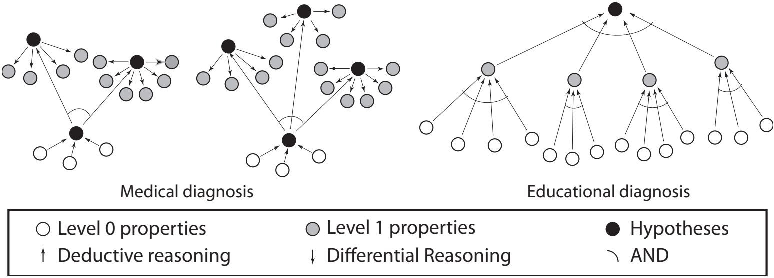


Figure 9.3: Graph describing the inference processes for medical diagnosis and educational diagnosis

defined according to the reasoning (it can be deductive or differential). Moreover, we say that the inference process is *static* if there is only one hypothesis to evaluate and the entities involved keep the same properties throughout the whole diagnostic process. However, if the properties of the entities change during the process and there is more than one hypothesis, we say that it is a *dynamic* process. This way, medical diagnosis is a dynamic process that requires differential reasoning (Fig. 9.3-left); but educational diagnosis is an static process which requires to apply deductive reasoning (Fig. 9.3-right) (Cabello and Ramos 2009). Thus, the base architecture for ESs in the medical diagnosis domain will have a *Differential Inference Engine* component. The behaviour of this component will differ from the behaviour of the inference engine of the ES in the educative diagnosis domain, which will have a *Deductive Inference Engine* component. Fig. 9.3 represents the medical and educative diagnosis processes as inference graphs.

9.2.4 *Diagnostic Expert Systems Application Domain Variability*

Variability management can not be completely achieved by using the domain feature model and the functional feature model. Some variability arises from the application domain: i. e., some products will share the same features and PRISMA software architecture. However, the PRISMA software architecture should be decorated with

different element according to the application domain, as the entities to diagnose will be different, and the rules and the hypothesis will vary too.

9.2.5 *Conclusions*

Based on the analysis of the diagnostic process carried out, we can conclude that:

- Diagnosis consists of an interpretation of the states of the involved entities (viewed as a set of properties), followed by the identification and specification of domain properties using rules.

- There is variability in the diagnostic process (system behavior) and this variability can be described in terms of its features as follows: (i) *Entity views*: an entity can be considered to participate with the same properties (the same view) or have different properties (different views) during the diagnostic process. (ii) *Property levels*: the properties of the entities can have n different abstraction levels. (iii) *Number of hypotheses*: the goal of the diagnosis is a single validated hypothesis, but there can be one or several candidate diagnostic hypotheses that must be evaluated in order to select the valid one. (iv) *Reasoning types*: these show the ways in which the rules are applied by the inference engine in order to infer a final diagnosis.

- There is also variability in the user requirements. We have elicited and modeled as UML Use Case diagrams (OMG 2010b) the user interaction requirements since they impact on the software architecture structure. We can describe this variability in terms of its features as follows: (i) *Number of use cases* of the system and how the system interacts with the environment (final users). (ii) *Number of actors*: number of final users of the

system. (iii) *Use cases per actor*: an actor can access different use cases.

- However, the features of the application domain must also be considered. Therefore, another variability emerges: the application domain variability. The features that correspond to a specific application domain are: (i) *Name* and *type* of the entities' properties by abstraction level. (ii) *Rules by abstraction levels* (the rules describe how the entity properties are related inter-levels). (iii) *Level*, *name* and *type* of the hypotheses used in the diagnostic process.

9.3 BOM INITIAL PROPOSAL: BOM–EAGER

The initial BOM proposal is called BOM–Eager.

BOM is a framework for variability elicitation, specification, and management in the Domain Engineering phase of a SPL. It is used for the enactment of the software production plan during the Product Engineering phase of the SPL.

9.3.1 *Variability management in BOM*

Given the classification of variability identified in the field studiy, the specification of the variability and system functionality is modeled in separate conceptual models in BOM. The user introduces the instances of these conceptual models in order to define the domain features used to produce the corresponding assets, or introduces to BOM the application domain features used to configure the final application, respectively.

The process of developing a specific application (member of the product line) begins with a domain-dependent generic architecture, which is unique. This implies that domain variability is captured in additional features that are represented as variants of the variability points. This variability is initially reflected in several base architectures (which are derived from the generic architecture). We also considered application domain variability to be necessary, and as a

result, we found that there are two types of orthogonal variability: the first one comes from the particular domain (diagnostic ES in our case study), and the second one comes from the application domain (for example, diagnostic medical ES).

For this reason, and based on the field analysis carried out, we conclude that the variants in ES are captured in two types of orthogonal variability which should be managed separately:

BOM proposes to manage two orthogonal sources of variability in two stages.

1. Variants of the domain, for example, the diagnosis, (i. e., the features of the domain associated with the behavior of the architectural elements), and the end-user requirements (i. e., features related to the structure of the architectural elements and the system itself)

2. Variants of the application domain. These variants are the features of the final products.

Fig. 9.4 presents a schema with features (variants) of these two variabilities. These two variabilities (domain variability and application variability) are managed in two stages. These stages correspond to the development of the SPL in two steps: the base architecture (SPL1), where a generic architecture is shared and the application in a specific domain (SPL2), which shares a base architecture.

BOM captures both variabilities in two models: the Domain Conceptual Model (DCM) and the ADCM, respectively.

9.3.1.1 *First variability management*

The *V1* variability (related to the features of the domain) is represented and managed through two models: the first model corresponds to the functional model of the domain, represented by the generic architecture of our SPL. This architecture is shared by several base architectures (or templates of base architectures), which represent the first SPL: *SPL1*. The second model is the model of the domain variability. Fig. 9.5 shows the original model used to describe the domain variability. As described in (Cabello Espinosa 2008), it uses

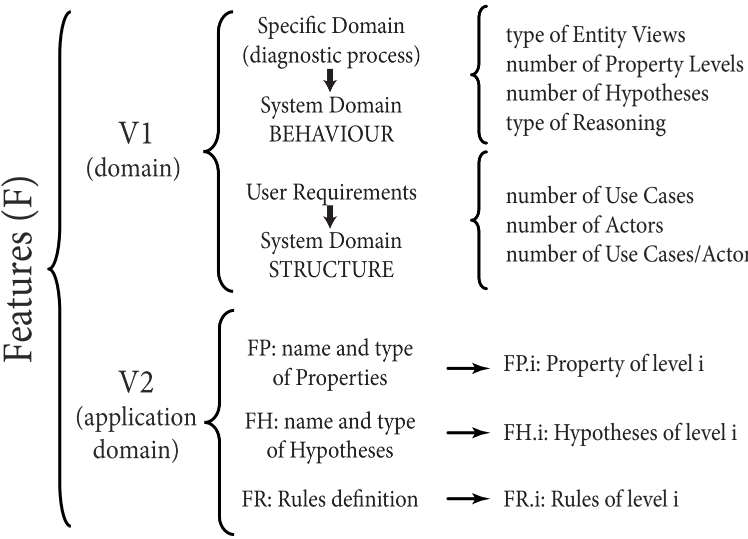


Figure 9.4: Features of the first variability (V1) and the second variability (V2) of our SPL

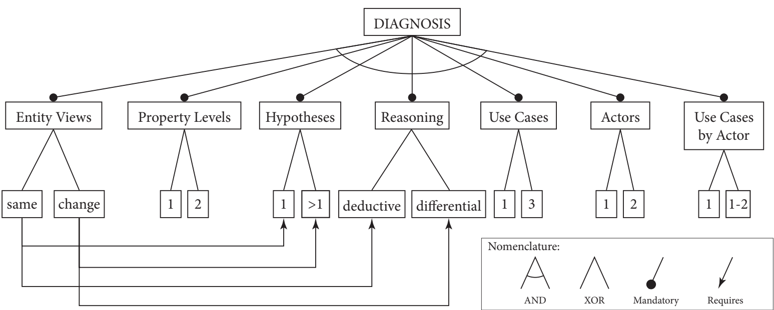


Figure 9.5: Original Feature Model of the first variability in BOM

an alternative notation to the traditional one, as it serves better for its purposes.

Based on the previous feature model a DCM was manually built as shown in Fig. 9.6. In this way, the features selected in the feature model are instances of the DCM. For example, the domain features in the medical diagnosis can be summarized as text as:

```
1 entity views = same
2 property levels = 2
3 hypotheses = 14
4 reasoning = differential
5 use cases = 3
6 actors = 2
7 use cases by actor = { actor_a = 2, actor_b = 1}
```

The instances of this model are created by the domain engineer capturing the first variability during the application engineering phase. These instances are used to produce the skeleton base architecture that corresponds to the choice made (see Fig. 9.8).

9.3.1.2 *Second variability in the BOM framework*

The second variability (*V2* variability) is also represented and managed using two models. The first model corresponds to the functional conceptual model of the application domain, which is captured by the skeleton base architecture of the *SPL1*. The second model corresponds to the feature model of the variability of the application domain, which is translated to the ADCM. The features of the application domain are used to instantiate the skeletons in order to obtain the PRISMA types (see Fig. 9.8).

It is important to mention that a skeleton base architecture can be instantiated into one or more PRISMA architectures. An example of

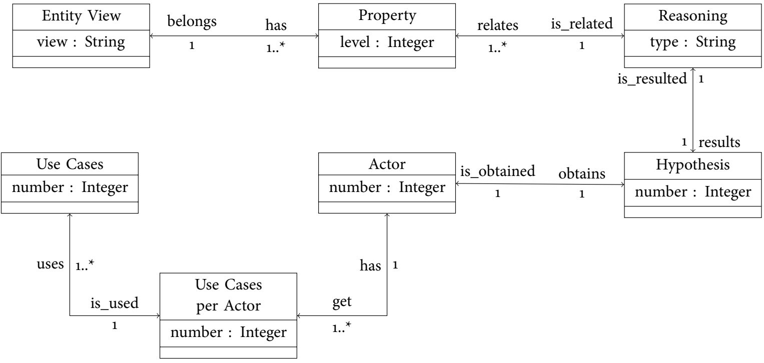


Figure 9.6: The domain conceptual model

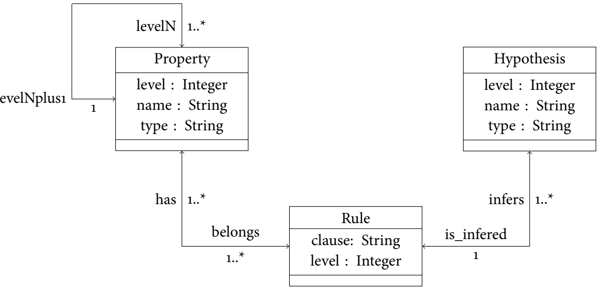


Figure 9.7: The application domain conceptual model

this situation appears in the two case studies: educative diagnosis and television diagnosis. These two cases share the same skeleton base architectures because they have the same variants of the first variability however they have different PRISMA architectures, because different properties of the two application domains are inserted in each skeleton base architecture.

The second type of variability involves the features of the application domain in a specific field, i. e., the SPL: *SPL2*. This variability allows the base architectures to be enriched or decorated with the application domain features.

In the application variability management process, the variations of the specific requirements of the application domain should be selected. This selection is reified as an ADCM instance given by the user. The features are inserted in the base skeletons in order to generate the types of the PRISMA software artifacts. These PRISMA architectural elements will be used to configure the PRISMA architectural model of the application.

The ADCM, which is shown in Fig. 9.7, captures the application domain variability. The instances of this model are created by the application engineer, and they capture the specific application domain variants. Some instances of this model for the diagnosis of infantile infectious diseases are shown in Listing 9.1.

Listing 9.1: Sample instances of the ADCM

```
1 properties of level 0: cough, fever
2 properties of level 1: dry_cough, constant_fever
3 hypotheses of level 1: warmth, parotiditis
4 hypotheses of level 2: pneumonia, bronchitis
5
6 rules: IF (cough=true and fever=true and
      respiretory_dificulty=tue) THEN syndrome=warth
```

9.3.2 *Software system views in BOM–Eager*

In BOM, two kinds of views for expert systems are considered: the *System Variability View* and the *System Functional View*.

The *System Variability View* is described using the two variability conceptual models: the DCM, which captures the domain and user variabilities (V1), and the ADCM capturing the application domain variability (V2). The DCM conforms to the *V1 Metamodel* (MM V1), and the ADCM conforms to the *V2 Metamodel* (MM V2). Both metamodels are the UML2 class diagram metamodel, but other domain specific metamodels can be used, producing other domain-specific models to capture the System Variability View.

The *System Functional View* is given during the production process by means of two views, which are described using three architectural models:

THE MODULAR VIEW for the Generic Architecture Model which conforms to the Modular Metamodel (MM Modular),

THE COMPONENT-CONNECTOR VIEW for the Base Architecture Models which conform to the Skeleton Metamodel (MM Skeleton) and the PRISMA Architecture Models, which conform to the PRISMA Metamodel (MM PRISMA) for the final product. The Skeleton Metamodel is similar to the PRISMA Metamodel, but it allows feature holders (holes).

9.3.3 *Relationships among system views*

As expressed previously, the variability is managed in two different phases in BOM. Such phases, identified as *SPL1* and *SPL2* involve different models and relationships among them.

Specifically, two types of relationships have been identified: (i) the relationships between the *Modular view* and the *Skeleton view*, (ii) the relations between the *Skeleton view* and the *PRISMA view* together with the views that define the SPL variability. Their respective profiles are:

$$R1: \quad \text{Modular}_{MM} \times \text{V1}_{MM} \times \text{Skeleton}_{MM}$$
$$R2: \quad \text{Skeleton}_{MM} \times \text{V2}_{MM} \times \text{PRISMA}_{MM}$$

Fig. 9.8 shows the process followed in BOM–Eager in the construction of a SPL architecture. This figure illustrates how the two types of varibility are considered in order to obtain the final product of the SPL.

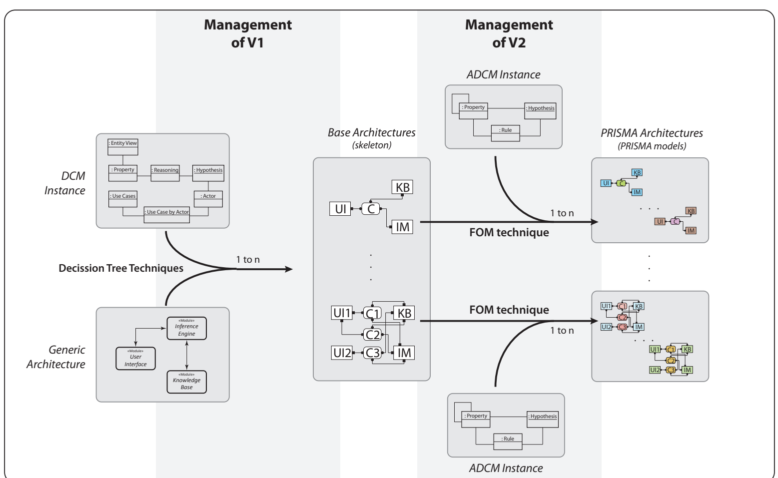


Figure 9.8: Variability management and system views in BOM–Eager

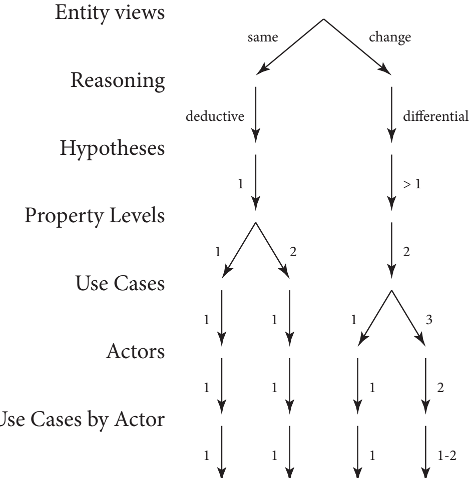


Figure 9.9: Binary Decision Tree to select a skeleton architecture for a DES

In the BOM–Eager approach, the different assets that participate in the SPL are developed at once and stored in the Baseline for each input in the domain engineering phase (see section 9.3.4.1).

In BOM–Eager, skeleton architectures are retrieved from a repository by using decision tree techniques.

In this way, given all the variants, the first variability is managed by an access to the repository that implements the Baseline. By means of decision tree techniques, we can select the base architecture of the specific case, given the variants of the variability points (features of the domain) as instances of the DCM. The variability points of the first variability are represented in the nodes of this decision tree, and its leaves represent the asset families of the SPL.

Fig. 9.9 shows the decision tree for the DES domain. This Binary Decision Tree (BDT) is constructed using the information of the feature model shown in Fig. 9.13. Given a configuration of the feature model, the decision tree selects the correct skeleton architecture from the baseline.

The second variability *V2* is managed using Feature Oriented Modeling (FOM) techniques. A decoration process of the base architectures is implemented employing the application domain features given as instances of the ADCM.

9.3.4 *Modeling the BOM approach*

BOM–Eager is based on two OMG standards: RAS, which identifies, describes, and packs assets in a standard way; and SPEM, which defines the standard language for modeling the software process. In BOM, a clear separation between the *domain engineering* and *application engineering* phases is made. This partition is the basis for reuse and automation of the software process. In the *domain engineering* phase, a set of assets/transformation rules and processes are created. In the *application engineering* phase, by executing the production plan, these assets are used/created to produce software products of high quality with a minimal cost and time.

In BOM, the domain engineer creates the production plan and all the software artifacts that are necessary to carry out the various tasks. The application engineer provides information of the application domain to the production plan process during its enactment, thus obtaining the final product.

9.3.4.1 *Domain engineering: building assets*

In the domain engineering phase as Computational Independent Model (CIM) and PIM all the software artifacts are built.

In BOM–Eager, all the core-assets are built and stored at the domain engineering stage in a repository called baseline.

The domain engineer creates the *baseline* as the repository of all the assets necessary to obtain a SPL product as Fig. 9.10 represents. The baseline is structured as a set of *Kit-Boxes* and the *production plan* (asset) of the SPL, where we store the assets and the know-how of how to use them in order to produce the SPL. The baseline and its assets are modeled at a high abstraction level.

The Kit-Boxes are packaged as a new composite asset, and they contain all assets and know-how for building a specific application and are the recovery units. A kit-box contains XML documents, processes, models, information, and configurations.

The Production Plan describes the process by means of tasks or activities to obtain a final product of the SPL. This process shows the production life cycle of our SPL. In the domain engineering phase,

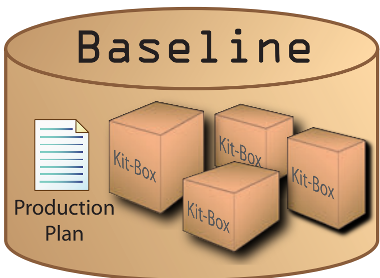


Figure 9.10: The baseline

this process is specified in SPEM, and it is inserted in the Baseline as an asset. In the application engineering phase, the Production Plan is recovered and enacted to generate an application as one SPL product.

9.3.4.2 *Application engineering: executing the Production Plan*

In the BOM–Eager approach, the user (application engineer) builds an application (a product of the SPL), by just giving as input the features of the variabilities *V1* and *V2* by means of instances of the DCM and ADCM conceptual models, respectively.

The Production Plan of our SPL in the BOM–Eager approach is shown in Figure 9.11 by using SPEM notation.

The Production Plan starts when BOM obtains (from the application engineer) the features expressed as DCM's instances of the variability points of the first variability. Next, BOM selects the assets from the Baseline, i. e., one Kit-Box asset that corresponds to the specific product. The Kit-Box that is selected by the engineer using BOM must be unpackaged in order for each asset to be used.

One of the assets recovered from the Kit-Box is the ADCM, which is used by BOM in order to obtain (from the application engineer) the features of the application domain considered as variants of the second variability.

Other assets recovered from the Kit-Box are the *Packaged Hybrids*. These assets are in turn unpacked to produce the PRISMA type

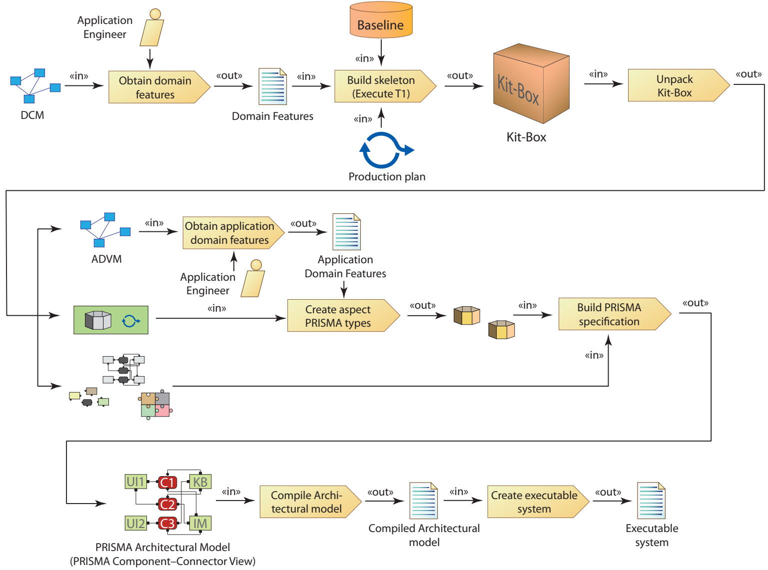


Figure 9.11: Production plan through BOM–Eager approach

aspects. BOM applies the Main Feature Insertion Process (which invokes the individual features insertion processes) to fill the selected skeletons (aspects) with the specific features of the case study defined by the engineer, thereby creating the PRISMA type aspects.

BOM uses the aspect PRISMA types and the other PRISMA type artifacts (interfaces architectural elements, and architectural model) to create the PRISMA configuration. These artifacts are input to the PRISMA-MODEL-COMPILER tool (Pérez et al. 2008) to automatically generate the code (in C# .NET). At the end, BOM creates the final system as an executable application, i. e., a final product of the SPL. This application is executed on top of the PRISMA-NET MIDDLEWARE (Pérez et al. 2008).

9.3.5 *BOM–Eager implementation*

The prototype for the BOM–Eager approach is called *ProtoBOM*. The Production Plan process diagram is used as a graphical metaphor for the user interface—GUI—in SPEM notation.

The initial BOM proposal is implemented in a prototype called ProtoBOM.

ProtoBOM can be used by the application engineer to generate a product of the SPL. *ProtoBOM* integrates the use of several tools, offering an approach to build applications, in a simple way. Some of the tools have been developed on purpose in *ProtoBOM* and others had already been created for other domains. The application engineer only introduces the variability features to the system in order to build an application (product of SPL). The rest of the activities will be carried out automatically by *ProtoBOM*.

In *ProtoBOM*, the *V1* variability is solved by means of our tool using decision tree techniques in order to select a Base Architecture. The *T2* transformation is executed by means of our tool using FOM techniques, in order to decorate this architecture. The baseline has been implemented as a repository accessed by a web service. In this way, the software product lines that are developed can be easily shared and distributed, which promotes the reuse of the assets contained in the baseline.

The *Asset Selection Process* computes paths in the decision tree which are used to select the assets in the baseline. This process is created by the domain engineer and uses the decision tree and the assets of the baseline. This process is executed by *ProtoBOM* in the application engineering phase, when the application domain engineer instantiates the DCM (i. e., inserting the specific domain information into the system as variants of the domain variability points). This process uses the decision tree techniques in order to carry out the *V1* resolution.

The *Feature Insertion Process* is used to insert the application domain features in their specific skeleton aspect. This process uses the FOM technique in order to carry out the *V2* resolution.

We explain how to insert the application domain features in a skeleton aspect in order to obtain the corresponding PRISMA type aspect.

The features are modeled as functions, representing refinements of the input model. In the educational program diagnosis, we have: $F_x.i \cdot$ *S-KB-ED*$_j$, which means "add feature $F_x.i$ to the *S-KB-ED*$_j$ model", where *S-KB-ED*$_0$ is the *Skeleton-Knowledge Base Educational Domain*, $0 \leqslant j \leqslant n$, and "$\cdot$" denotes the application of the function. The gluing process is iterated step by step:

$$S\text{-}KB\text{-}ED_{j+1} = F_x.i \cdot S\text{-}KB\text{-}ED_j$$

In Table 9.1, an example of the functional aspect skeleton and its respective aspect type for the Knowledge Base component of the educational program diagnosis is presented.

These software artifacts are XML documents that are specified using the PRISMA-Architecture Description Language (ADL). In order to simplify this example, we have omitted the XML syntax. In this Table, $\langle F_x.i \rangle$ are the features place holders. These features are the following: $F_P.i$ = Features of the properties of level i, F_H = Features of the hypothesis, $F_R.i$ = Features of the rules of level i.

We have used the PRISMA-MODEL-COMPILER to automatically generate C# code from the system architectural models and to create the application (an instance of the SPL), which is executable over the PRISMA-NET-MIDDLEWARE.

An example of (part of) the generated C# code (by the PRISMA-MODEL-COMPILER) that corresponds to the *Knowledge Base* com-

Listing 9.2: Example code generated by the PRISMA-MODEL-COMPILER

```
1 namespace KBMD {
2   [Serializable]
3   public class KnowledgeBaseEducational : ComponentBase {
4     public class KnowledgeBaseEducationalDiag string name :
          base(name) {
5       AddAspect(new FBaseED());
6       InPorts.Add("KnowPort", "IDomainED", "KNOW");
7       OutPorts.Add("KnowPort", "IDomainED", "KNOW");
8     }
9   }
10 }
```

FUNCTIONAL ASPECT OF THE KNOWLEDGE BASE OF A SKELETON	FUNCTIONAL ASPECT OF A PRISMA TYPE
Functional Aspect FBase using IDomain	Functional Aspect FBase using IDomain
Attributes	Attributes
Variables	Variables
$\langle$ F_P.0 $\rangle$	pictures: string;
	TVstructure: string;
	music: string;
	...
Deriveds	Deriveds
$\langle$ F_P.1 $\rangle$	production: string;
	content: string;
$\langle$ F_H $\rangle$	technicalQuality: string;
	stateVideo: string;
Derivations	Derivations
$\langle$ F_R.1 $\rangle$	{pictures="proper" and TVstructure="authorized" and
	music="good"} production:="good";
$\langle$ F_R.2 $\rangle$	...
	{production="good" and content ="good" and
...	technicalQuality="good"} stateVideo:="VideoOK";
	...
Services	Services
...	...
End Functional Aspect FBase	End Functional Aspect FBase

Table 9.1: Example of a skeleton and PRISMA type aspects

ponent of the educational program diagnosis case study is shown in the Listing 9.2.

9.4 TURNING BOM INTO A MMDSPLE PROCESS: THE BOM–LAZY PROPOSAL

The BOM–Eager approach uses models to describe the variability and the systems' structure, however, these models are not reused in a complex MDE process. The BOM–Lazy approach aims to turn BOM–Eager into a MMDPLE process.

The initial proposal of BOM, BOM–Eager, proposes a simple but powerful framework for the development of Software Product Line (SPL). In BOM, variability is managed in two different stages, first, the variability of the domain is taken into account; and second, the variability of the application domain is considered. The BOM–Eager proposal is adequate qhen the baseline size is small, as all the assets are built and stored extensively during the *domain engineering* phase. Moreover, decision tree techniques drive the *V1* management and Feature Oriented Programming (FOP) techniques are used to manage the *V2* variability.

However, when the size of the baseline is huge this proposal becomes inefficient. The BOM–Lazy approach arises as the solution to this problem. The BOM framework deals with several models (V1 and V2 variability models, modular model, skeleton models and architectural models). This way, a multi-model driven approach can ease the use of different models and systems views, and can provide the tools to implement a production plan driven by model transformations.

This section describes how the different BOM models have been adapted to the MOF architecture, and how the production plan has been adapted to use multi-models. This way, the relationships among system views (which in BOM where managed using BDT and FOM techniques) are now expressed using declarative QVT-Relations rules.

9.4.1 *Representing the first variability in BOM–Lazy*

In BOM–Lazy, the *V1* variability is also represented and managed through two models: the functional model and the domain variabil-

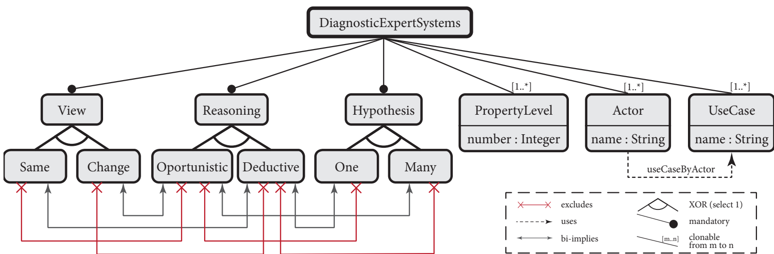


Figure 9.12: Feature Model of the first variability in BOM

ity model. This model is represented by a feature model as shown in Fig. 9.12.

This feature model is an adaptation of the original feature model to the cardinality-based feature modeling notation. As can be seen, it makes use of feature multiplicities and feature attributes. To make possible the use of such feature model in a complex MDE process, this feature model is automatically translated to the equivalent DVM as explained in section 7.3.

In the context of BOM, the DVM of the first variability corresponds to the DCM following the naming conventions used in BOM–Eager. The automatically obtained DCM is shown in Fig. 9.13. This way, the features selected in the feature model are instances of the DCM as in the BOM–Eager approach happens. However, in BOM–Lazy we avoid the manual creation of the DCM as it is automatically obtained.

As in the initial proposal, the instances of the DCM are used to produce the skeleton base architecture.

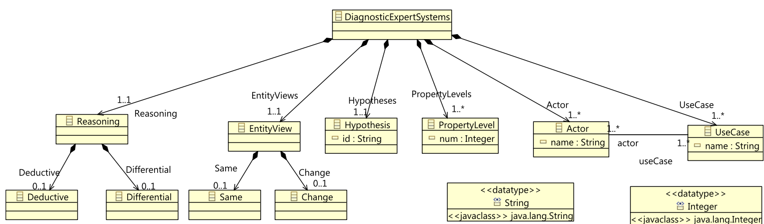


Figure 9.13: The domain conceptual model

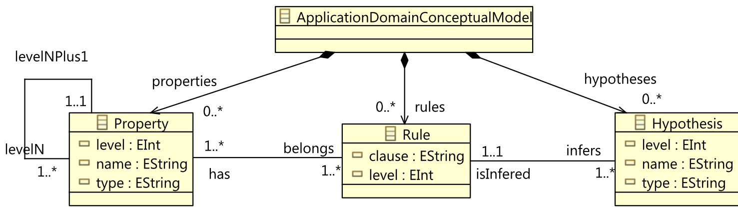


Figure 9.14: The application domain conceptual model

9.4.1.1 *Representing the second variability in BOM–Lazy*

The second variability (V2) is represented using two models as in the BOM–Eager approach. The first model corresponds to the skeleton architecture obtained from the first variability management phase (SPL1). The second model corresponds to the application domain variability model. This model is used to obtain the ADCM. However, the ADCM is automatically obtained in BOM–Lazy; as it occurs in the case of the DCM. The instances of the ADCM are used to decorate the skeleton architectures with the application domain features. Nevertheless, in the BOM–Lazy approach the process to enrich the skeleton architectures is performed by using model transformations instead of FOM techniques as we will describe un section 9.4.3.

The automatically obtained ADCM is shown in Fig. 9.14. It captures the application domain variability. The instances of this model are created by the application engineer, and they capture the specific application domain variants as in the case of BOM–Eager.

9.4.2 *Software system views in BOM–Lazy*

In BOM two kinds of views are considered for expert systems: the *System Variability View* and the *System Functional View*.

On the one hand, the *System Variability View* is described using the two variability conceptual models (DCM and ADCM) which conform to the UML2 class diagram metamodel.

On the other hand, the *System Functional View* is described using three architectural models:

- The *Generic Architecture Model*, which conforms to the Modular Metamodel (MM Modular).
- The *Base Architecture Model* which conform to the Skeleton Metamodel (MM Skeleton).
- The *PRISMA Architecture Model*, which conform to the PRISMA Metamodel (MM PRISMA).

We have used the MULTIPLE built-in metamodels based on the Modular and Component-Connector view types proposed by Shaw and Clements (2006) and Limón Cordero (2010).

Fig. 8.19 (see page 147) shows the Modular View Metamodel (MM Modular view). The main element considered for this view is the module itself. This figure shows that a model contains a set of modules (which can contain different functions), which are linked to other modules by means of relations (decomposition, uses and layer). The more relevant relation is the Use relation, although other types have been considered. The labels in the links are useful for indicating how the relation is made.

Fig. 8.25 (page 155) shows the Component-Connector View Metamodel (MM Component-Connector view). The component class and connector class are the main elements. Both are derived from a more general component class (TComponent). The components provide a set of services through a set of ports. The connectors link the ports of components by means of their roles. Different types of *relations* can be also defined between *components* and *connectors*.

9.4.3 *Relationships among metamodels*

We used QVT to define relationships among metamodels in MOF. Specifically, the QVT-Relations language is used to describe them.

The source and target metamodels are identified first. The correspondence among each element of the metamodels must be defined

In BOM–Lazy, relationships among the different views of a system are described by using QVT–Relations. Thus, it is possible to specify in a declarative way how the behaviour and the structure of a system may vary according to a feature model configuration.

taking into account the way in which their rules are represented through QVT. In this case, the source metamodels are the *Modular metamodel* and the *V1 variability metamodel*. The target metamodel is the *Skeleton metamodel*. The rules considered in the relationships for the elements of the modular and V1 variability metamodels are *checkonly* type (to verify the elements) in the left part and *enforce* type in the right part to create the elements of the skeleton metamodel.

Fig. 9.15 shows the transformations involved in the construction of our SPL architecture. This figure is an adaptation of the process shown in Fig. 9.8 to the MMDPLE approach. It illustrates how the model transformations performed at the model level (M1) are applied to refine the SPL assets until the final PSM is obtained.

The modular model is made up of three modules: *InferenceEngine*, *KnowledgeBase*, and *UserInterface*. These are all linked with each other through dependence relationships. Next, the transformation task for producing the skeleton model (target) is performed. To

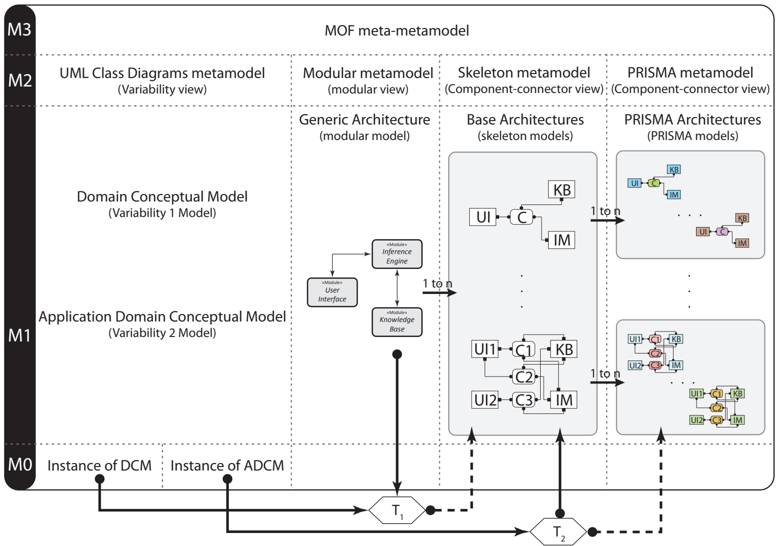


Figure 9.15: The T1 and T2 model transformations in BOM (located in the MOF levels)

do this, an instance of the DCM is used. A skeleton component is produced for each module, and each dependence relationship generates a skeleton connector.

The MOF level M1 is where the *T1* and *T2* transformations are carried out. A first model (skeleton model) is obtained by the *T1* transformation, using the modular model and a DCM instance as sources. This skeleton model is completed by means of the *T2* transformation by using an ADCM instance which allows the PRISMA model to be obtained as a refinement. The *T1* and *T2* model transformations are PIM to PIM transformations.

In the *T1* transformation, QVT-Relations takes into account the generic architecture, the instance of conceptual model of the domain variability, and the skeleton base architecture configuration. The *T1* transformation profile is:

$$T1(GenArch_{model}, V_1 model_{instance}) = BaseArch_{model}$$

In *T2* transformation, QVT-Relations consider the base architectures, the instance of the application domain variability, and the PRISMA architecture configuration. The *T2* transformation profile is:

$$T2(BaseArch_{model}, V_2 model_{instance}) = PRISMA_{model}$$

9.4.3.1 *BOM–Lazy production plan*

In the BOM–Lazy approach, the application engineer builds a product of the SPL, by just giving as input the features of the variabilities *V1* and *V2* by means of instances of the DCM and ADCM conceptual models, respectively.

The Production Plan of the SPL taking the BOM–Lazy approach is shown in Fig. 9.16 by using SPEM notation. The Production Plan starts (1) when BOM obtains (from the application engineer) the features expressed as DCM's instances of the first variability. Next, (2) BOM executes the QVT-Relations transformation *T1* and obtains the base architecture that corresponds to the specific product. Then,

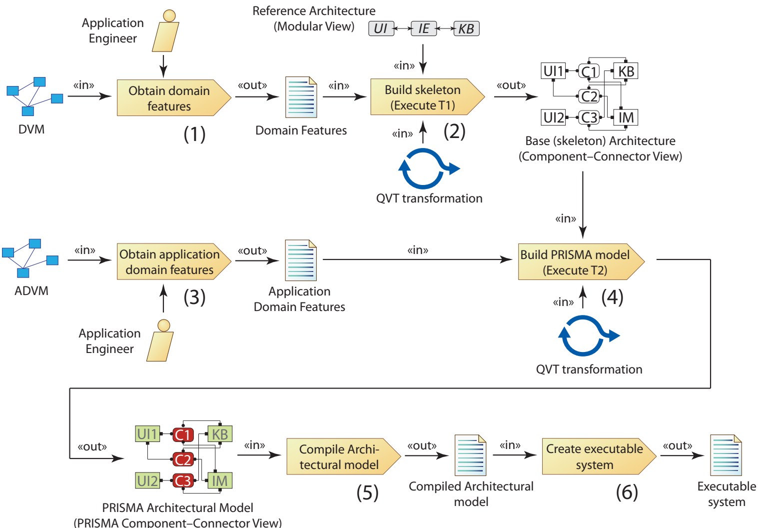


Figure 9.16: Production plan through BOM–Lazy approach

(3) ADCM is used by BOM in order to obtain (from the application engineer) the features of the application domain considered as variants of the second variability. Next, (4) BOM executes the QVT-Relations transformation *T2* and produces the PRISMA architectural model. This PRISMA type artifact is sent to the PRISMA-MODEL-COMPILER tool (5) , and the process continues as the BOM–Eager approach does.

9.5 BOM–LAZY IMPLEMENTATION

The BOM-Lazy architecture is shown in Fig. 9.17. In this approach, the model transformations *T1* and *T2* are executed by means of our tool using the QVT-Relations language.

The tool is built on top of the Eclipse platform. It uses EMF as the reference implementation of the MOF standard. The tool is made up by a set of graphical editors (used to define the models that are part of the SPL) and a QVT-Relations transformations engine.

Next, the transformations which specify the correspondences among the different models of the SPL are described. These transformations are the core implementation of the production plan.

9.5.1 *T1 transformation*

As presented in section 9.4.3, the transformation in charge of creating the base architectures of the SPL is called *T1*. This transformation is determined by the first variability *V1*. Thus, this transformation takes as inputs the model of the generic architecture, which describes the common parts of the ES (i. e., the *Modular Model*), and an instance of the variability model that captures the variability of the specific domain (i. e., instance of the DCM).

In order to increase the quality of the architectural design obtained, we will follow good software design practices as defined by Pressman (2001). Such good design practices are encoded as a set of patterns in the definition of the QVT rules, and will be applied to calculate the base architectures. As expressed before, the signature of the transformation is:

$$T1(GenArch_{model}, V_1 model_{instance}) = BaseArch_{model}$$

As opposed to the BOM–Eager approach, where the baseline stores all the knowledge about the base architectures in an explicit way, in BOM–Lazy all the knowledge is stored implicitly. Such knowledge is encoded by using a set of rules which describe how both

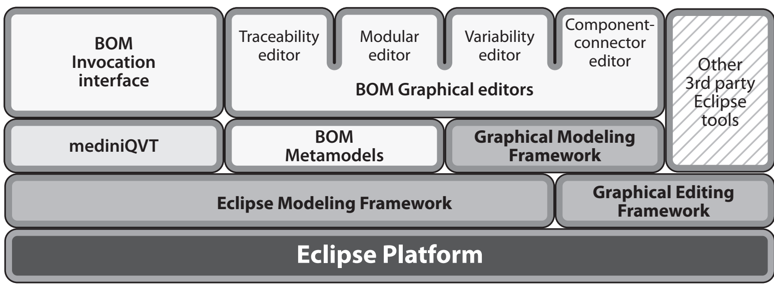


Figure 9.17: The BOM–Lazy architecture

system requirements (system variability view) and design patterns (good practices) are interrelated. This way, the creation of the base architectures is moved from the domain engineering to the application engineering phase. Each architecture is calculated just in the moment when it is needed, and not before.

Next, the QVT-Relations rules of the T1 transformation together with the good design pattern they satisfy are specified.

9.5.1.1 *Design patterns and quality guidelines*

Base architectures in BOM–Lazy can be created considering good design patterns.

A pattern, *is a three-part rule, which expresses a relation between a certain context, a problem, and a solution* (Alexander 1977). In this section we detail the patterns and guidelines that will drive the *T1* transformation. Pressman (2001) enumerates the following quality guidelines to achieve a good design:

1. *"A design should exhibit an architecture that (a) has been created using recognizable architectural styles or patterns, (b) is composed of components that exhibit good design characteristics [...], and (c) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.*
2. *"A design should be modular; that is, the software should be logically partitioned into elements or subsystems*
3. *"A design should contain distinct representations of data, architecture, interfaces, and components.*
4. *"A design should lead to data structures that are appropiate for the classes to be implemented and are drawn from recognizable data patterns.*
5. *"A design should lead to components that exhibit independent functional characteristics.*
6. *"A design should lead to interfaces that reduce the complexity of connections between components and with the external environment.*

7. *"A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.*
8. *"A design should be represented using a notation that effectively communicates its meaning."*

Moreover, Pressman (*op cit.*) suggests to classify the design elements (specifically design classes in the literature) in five different groups:

1. *User interface classes* — define the abstractions that are necessary for Human-Computer Interaction (HCI) interaction.
2. *Business domain classes* — identify the attributes and methods that are required to implement some elements of the business domain.
3. *Process classes* — implement lower-level business abstractions required to fully manage the business domain classes.
4. *Persistent classes* — represent data stores.
5. *System classes* — implement software management and control functions.

Given this classification of software artifacts and quality guidelines, the following design decisions were made. These design decisions will be later encoded as QVT-Relations object templates, which, in turn, specify a set of transformation patterns.

1. *Hierarcical organization*. As specified before, a design must define a hierarchical organization that controls software components in a smart way. In order to satisfy this criterion, the modular metamodel contains the ModulesModel element, which contains an organizes the rest of the elements which take part on it. In a similar way, the Component–Conector Metamodel contains the CCModel element which plays the same role.

This way, there is a one–to–one relationship between both ModulesModel and CCModel elements.

2. *Connectors*. Use cases constitute a partition of the system based on functionality. Thus, taking into account that a design should be modular and that software should be logically partitioned; for each use case we will create a connector in charge of coordinating the different components of the use case. This way, there is a one-to-one relationship between use cases and connectors.

   The number of connectors in the skeleton architecture will be equals to the number of use cases found defined in the *V1* instance model.

3. *User interfaces*. The reference architecture of ES is usually conformed by three modules: *Knowledge Base*, *Inference Engine* and *User Interface*. The *Knowledge Base* and *Inference Engine* modules lead to the *Knowledge Base* and *Inference Engine* components respectively. However, with the aim of reducing the complexity of connections between modules and with the external environment, the *User Interface* module is transformed into as many components as actors appear in the domain variability model (*V1*).

   This way, a one–to–one mapping is defined between the actors of the *V1* instance model and the user interfaces in the target model. As a result, the number of *User Interface* components will be equals to the number of actors which interact with the system.

4. *Uniqueness of the Knowledge Base*. Each use case generates an architectural model composed of three components: *Knowledge Base*, *Inference Engine* and *User Interface*, coordinated by a coordinator connector. We define as a design criteria the uniqueness of the *Knowledge Base* component, because knowledge must be unique for the entire system. For each use case there is a different view of the *Knowledge Base*. This

is accomplished by adding a port to the *Knowledge Base* that binds each connector with a different role. The *Knowledge Base* will have as many ports as views, i. e., it will have as many ports as use cases. A many–to–one relationship between use cases and the *Knowledge Base*. The number of ports in the *Knowledge Base* shall be equal to the number of existing use cases in the instance of the *V1* domain variability model.

5. *Uniqueness of the Inference Engine*. The type of reasoning in the diagnostic process is given by the *Reasoning* variant in the *V1* domain variability model. *Reasoning* can be defined as *Deductive* or *Differential* and will determine the type inference engine to use. The inference process can be deductive or differential but not both simultaneously. For each use case there will be a different view of the *Inference Engine*. This will be achieved by adding the needed ports to the *Inference Engine*. Such ports will be linked to roles of the corresponding connectors.

   The Inference Engine will have as many ports as views, i. e. it, will have as many ports as use cases we find in the DCM. A many–to–one relationship is defined between use cases and the *Inference Engine*. The number of ports of the *Inference Engine* will be equal to the number of existing use cases in the instance of the *V1* domain variability model.

6. *Interaction pattern*. Previously we specified that for each actor a *User Interface* component will be used. However, an actor can access several use cases. Thus, as the functionality of accessed by an actor can be partitioned, the same division will be done in the UI component. The different functionality of the *User Interface* component will be accessed by different ports. Thus, the number of ports of each UI component is equal to the number of use cases that the user can access.

7. *Connectors merge* This pattern extends the semantics defined by the pattern number 2. The existence of an *includes* relation-

ship between two use cases implies the merger of connectors. An included use case describes a subprocess of its parent. The number of connectors is equal to the number of top-level use cases in the instance of the *V1* domain variability model.

9.5.1.2 *QVT Rules*

The design decisions and guidelines described in the previous section are encoded using QVT. This allows us to specify them in an unambiguous way. They are visually described using the graphical notation of the QVT–Relations language. The metamodels involved in the transformation process are the ones described in section 9.4.2. The whole code of the transformation can be found in the Appendix B.

MODULESMODEL2COMPONENTSMODEL RELATION Fig. 9.18 shows the *ModulesModel2ComponentsModel* relation. This relationship is *top-level*, and transforms the ModulesModel element of the modular metamodel to the CCModel in the Component-Connector metamodel. The rules assigns to the new element the name of the source element. The `where` clause invokes the *UseCaseToConnector* relation. This rule codifies the pattern number 1.

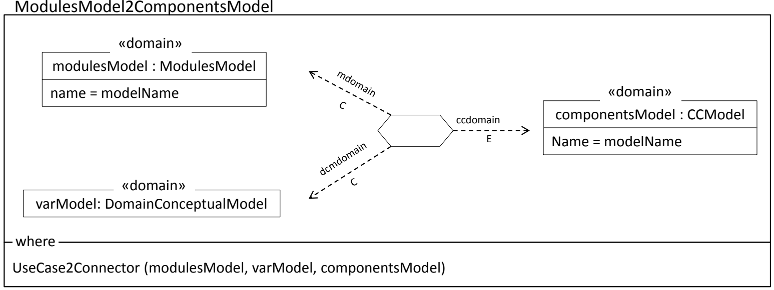


Figure 9.18: *ModulesModel2ComponentsModel* relation

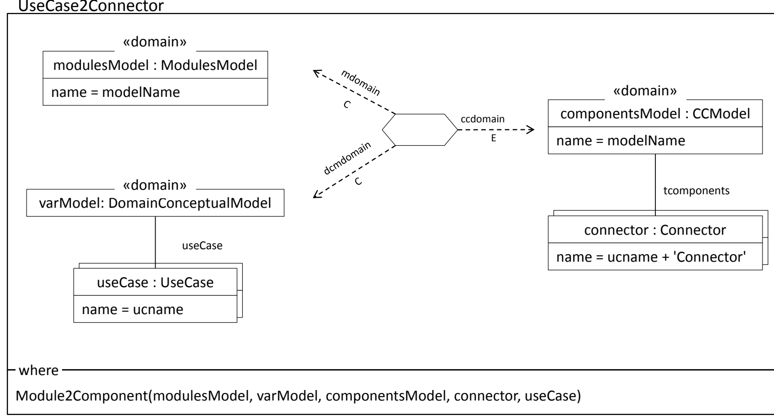


Figure 9.19: *UseCase2Connector* relation

USECASE2CONNECTOR RELATION Fig. 9.19 shows the *UseCase2Connector* rule using the graphical notation of the QVT language. This rule creates, for each one of the use cases of the source variability model one connector of the component–connector model. The name of such connector stands for the concatenation of the use case name and the suffix ''Connector''. Finally, the `where` clause states that the *Module2Component* relation must be considered as the post-condition of the current rule (*UseCase2Connector*). Thus, this rule should be properly applied after the *UseCase2Connector* relation is checked. This rule describes the pattern number 2.

MODULE2COMPONENT RELATION This rule (Fig. 9.20) creates, for each module of the modular model, a component in the component-connector model. The name of the new component is calculated using the *getComponentName* OCL query (see listing 9.3). If the module to transform is the *User Interface* module, the component name is formed by the module name and the name of the actor that accesses the use case related with the module. If the module is the *Inference Engine* or the *Knowledge Base*, the component name also reflects the type of reasoning (deductive or differential) specified by *Reasoning* variant selected in the *V1* variability model. Finally the

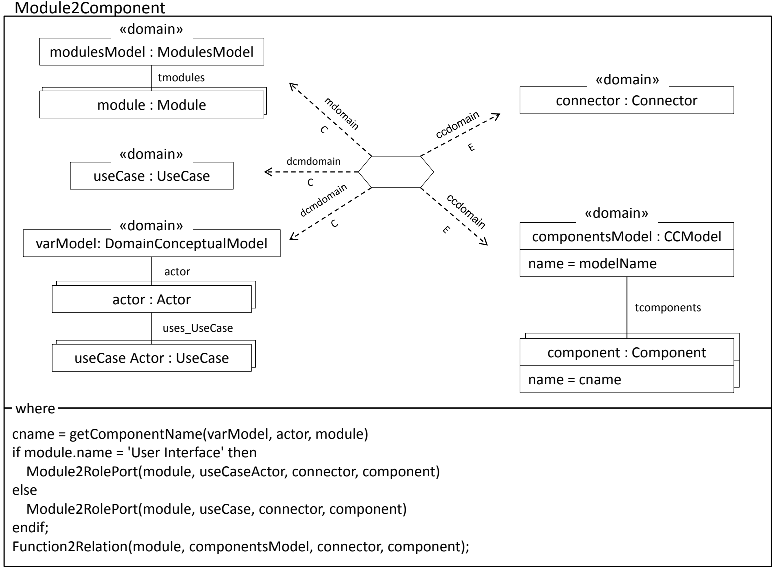


Figure 9.20: *Module2Component* relation

`where` clause states that the *Module2RolePort* rule should be invoked if the rule is applied over the *User Interface* module. In such case, the rule takes as arguments the module and the use cases accesses by the actor related with it (as the pattern number 6 describes). In any other case, the *Module2RolePort* rule is invoked with the use case related with the module as the patterns 3, 4 and 5 specify. Finally, the *Funtion2Relation* rule is invoked.

MODULE2ROLEPORT RELATION The criterion *Uniqueness of the knowledge base* indicates that knowledge must be unique for the entire system. In order to meet this requirement, there is only one *Knowledge Base* component in our base architecture. Given that for each use case exists a different view of the *Knowledge Base*, we must merge each one of these views. The *Module2RolePort* (see Fig. 9.21) relation is in charge of creating the necessary roles and ports in the components and connectors of the target metamodel (as the should not be duplicated if it is not necessary). This is represented

Listing 9.3: getComponentName(...) OCL query

```
1 query getComponentName(varModel : dcm::DomainConceptualModel,
      actor : dcm::Actor, module : mview::Module) : String {
2
3   if module.name = 'User Interface' then
4     module.name + ' - ' + actor.name
5   else
6     if module.name = 'Inference Motor' or module.name = '
          Knowledge Base'  then
7       if varModel.Reasoning.ReasoningFeatures.oclIsTypeOf(dcm
            ::deductive) then
8         'Deductive ' + module.name
9       else
10        if varModel.Reasoning.ReasoningFeatures.oclIsTypeOf(
              dcm::differential) then
11          'Differential ' + module.name
12        else
13          'Error ' + module.name
14        endif
15      endif
16    else
17      module.name
18    endif
19  endif
20 }
```

in the *Knowledge Base* component by adding a port that allows the communication of such views with the rest of the components. To achieve this, the name of the port is composed by the use case name and the ‘‘Port’’ suffix.

To allow the communication among views, it is also necessary to add the corresponding roles to the adequate connectors. The name of the roles is composed by the name of the module plus the ‘‘Role’’ suffix. This way, a one-to-one relationship is defined between use cases and the *Knowledge Base* ports, which also implies a one-to-one relationship between use cases and the roles of the different connectors. This rule also creates the corresponding ports in the *Inference Engine* and *User Interfaces* components, and their corresponding roles in the involved connectors. The `where` clause indicates that the *ConnectRoleAndPort* and *Function2Service* relations should be con-

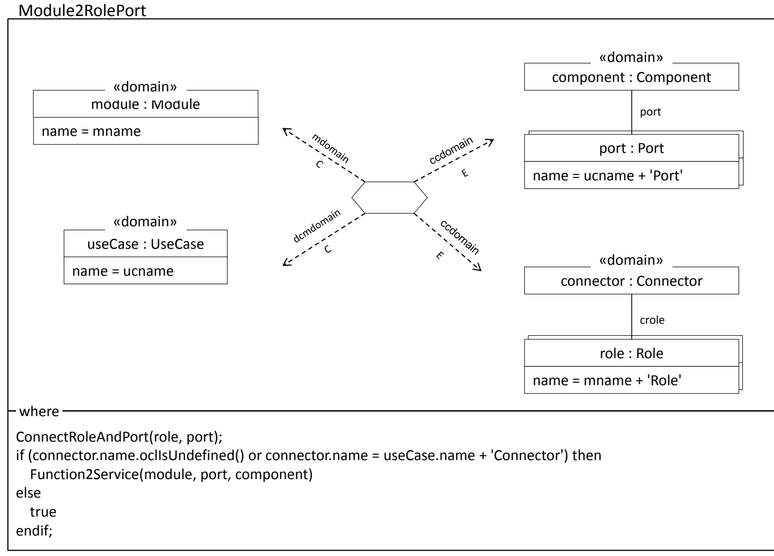


Figure 9.21: *Module2RolePort* relation

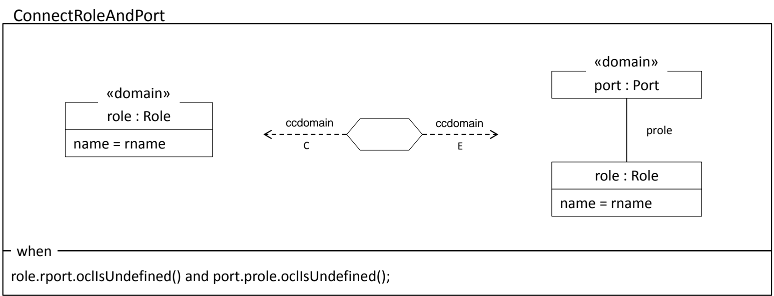


Figure 9.22: *ConnectRoleAndPort* relation

sidered as post-conditions. The *Module2RolePort* relation represents the patterns 4, 5 and 6.

CONNECTROLEANDPORT RELATION This rule can be considered an utility rule. *ConnectRoleAndPort* (Fig. 9.22) relates elements of the Component–connector metamodel. Specifically, it establishes the link between the given Role and the given Port.

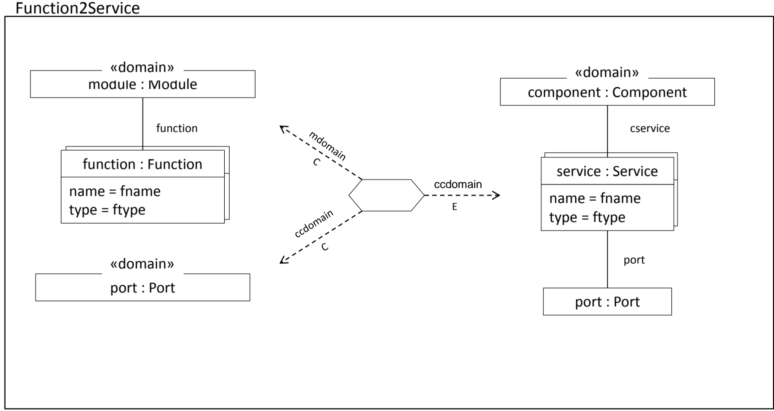


Figure 9.23: *Function2Service* relation

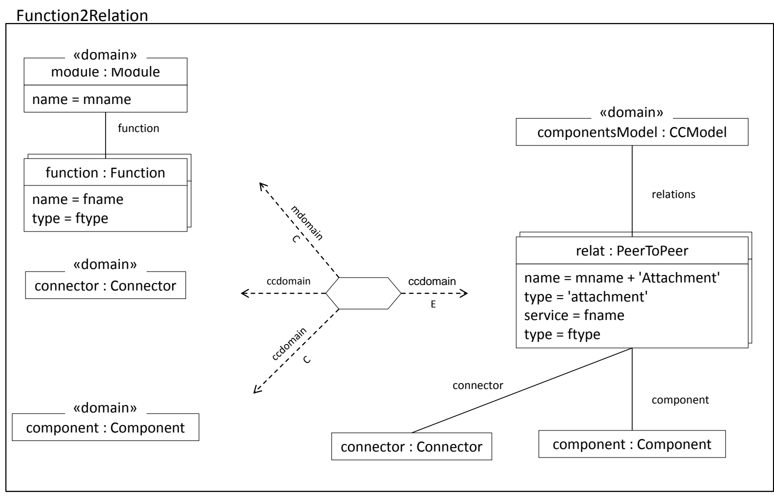


Figure 9.24: *Function2Relation* relation

FUNCTION2SERVICE RELATION Fig. 9.23 describes the *Function2Service* relation. This rule creates a new service on the corresponding component of the Component–Connector model. This service is generated from a given function of a module of the modular model. The new service is created with the same name and type of the source function.

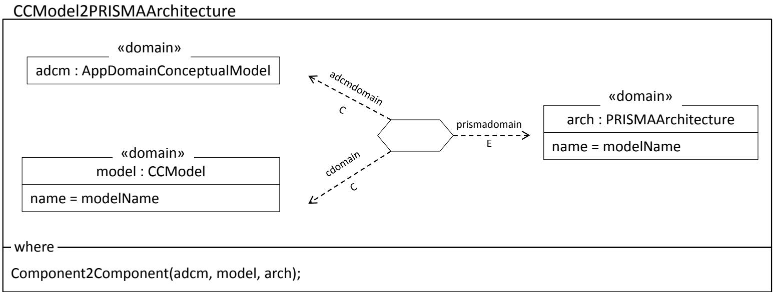


Figure 9.25: *CCModel2PRISMAArchitecture* relation

FUNCTION2RELATION RELATION The rule represented in Fig. 9.24 states that a function of a module of the source model (ModuleModel) will generate a Relation element in the target model (CCModel). This Relation will link a connector with its corresponding component in the Component–Connector model.

9.5.2 *T2 transformation*

The *T2* transformation is the last step before obtaining the executable application. Three domains participate in the transformation: the *ccview* domain (the skeleton Component–Connector metamodel), the *adcmdomain* (the *V2* application domain variability model) and the *prismadomain* (the PRISMA architectural model). The *CCModel2PRISMAArchitecture* rule is top-level and will be the first relation to be invoked. The other rules are non-top-level and will be invoked as post-conditions of other rules.

Next, all the rules of the T2 transformation are shown and described in detail. The code of the transformation can be found in the Appendix C.

9.5.2.1 *QVT Rules*

CCMODEL2PRISMAARCHITECTURE RELATION The *CCModel2PRISMAArchitecture* rule shown in Fig. 9.25 transform a CC-

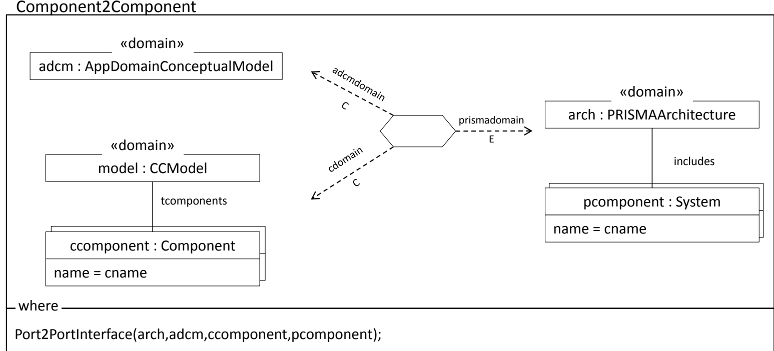


Figure 9.26: *Component2Component* relation

Model element to a PRISMAArchitecture element, and assigns to it the same name. The CCModel element is the one that contains the rest of the *Component-Conector model* elements. The PRISMAArchitecture element will play the same role, but in the *prismadomain* domain.

COMPONENT2COMPONENT RELATION The relation shown in Fig. 9.26 describes the *Component2Component* relation. It transforms each component in the source domain to a component in the target domain, and assigns to it the same name.

PORT2PORTINTERFACE RELATION The *Port2PortInterface* rule (see Fig. 9.27) transforms each port of a component in the *Component–Connector metamodel* to the corresponding port in a component of the PRISMA domain. The type of the new port es defined by creating the corresponding interface in it.

COMPONENT2FUNCTIONALASPECT RELATION Fig. 9.28 shows the *Component2FunctionalAspect* rule. It creates a functional aspect in each PRISMA component. It also transforms each component service in a PRISMA service with the same name and type; and assigns it to the new aspect of the PRISMA component. It also assigns the set of interfaces that the aspect implements. The `where`

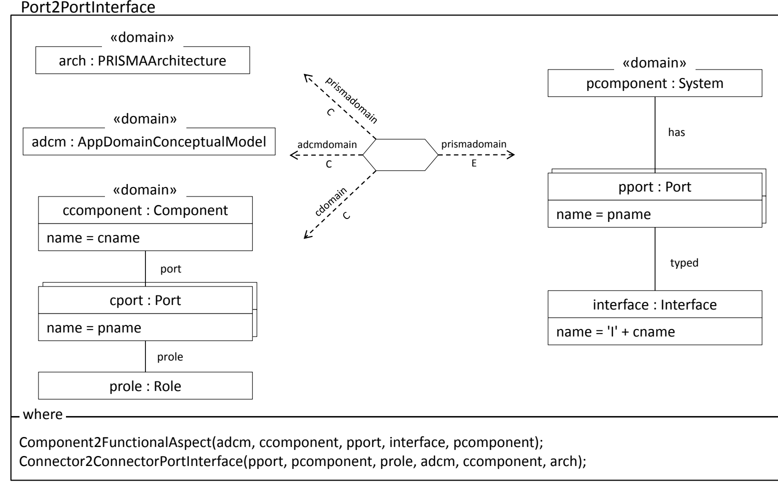


Figure 9.27: *Port2PortInterface* relation

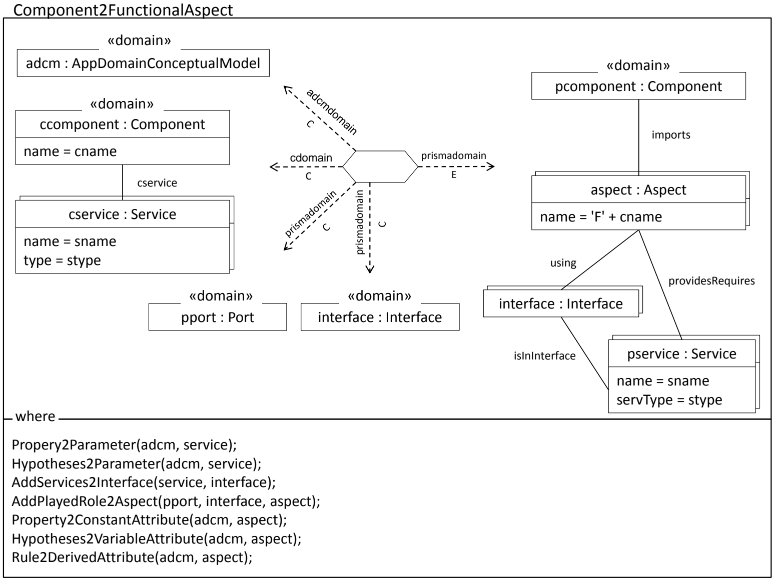


Figure 9.28: *Component2FunctionalAspect* relation

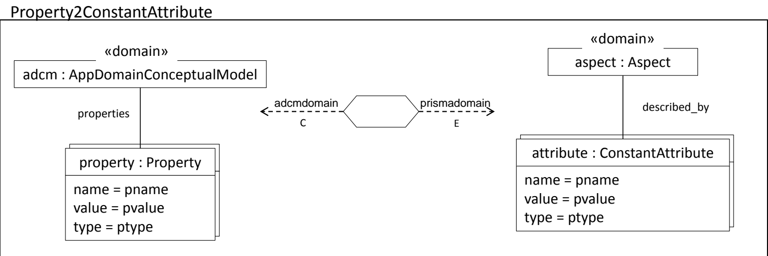


Figure 9.29: *Property2ConstantAttribute* relation

clause invokes the following rules: *AddServices2Interface* to assign the public services to the port interfaces; *AddPlayedRole2Aspect* to create the played_role in the aspect of its corresponding interface (the one which defines its behavior). This rule in turn invokes the *AddPlayedRole2Port* rule, which assigns the played_role to the appropriate port. The *Property2ConstantAttribute*, *Hypotheses2VariableAttribute*, *Rule2DerivedAttribute* rules are also invoked to create the attributes of the PRISMA aspect. Such rules are explained next.

PROPERTY2CONSTANTATTRIBUTE RELATION Fig. 9.29 shows the *Property2ConstantAttribute* relation. It transforms each property of the ADCM in a constant attribute of a PRISMA aspect. The rule assigns to the attribute the same name and type.

HYPOTHESES2VARIABLEATTRIBUTE RELATION Fig. 9.30 shows the *Hypotheses2VariableAttribute* relation. It transforms each hypothesis of the ADCM in a variable attribute of a PRISMA aspect. The rule assigns to it the same name and type.

RULE2DERIVEDATTRIBUTE RELATION The *Rule2DerivedAttribute* relation (see Fig. 9.31) transforms each rule of the ADCM in a derived attribute of a PRISMA aspect. The relation assigns the clause of the rule as its term.

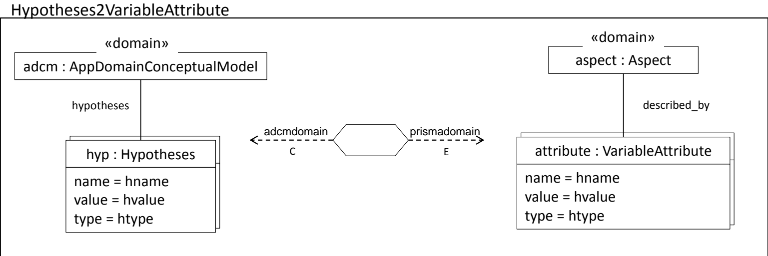


Figure 9.30: *Hypotheses2VariableAttribute* relation

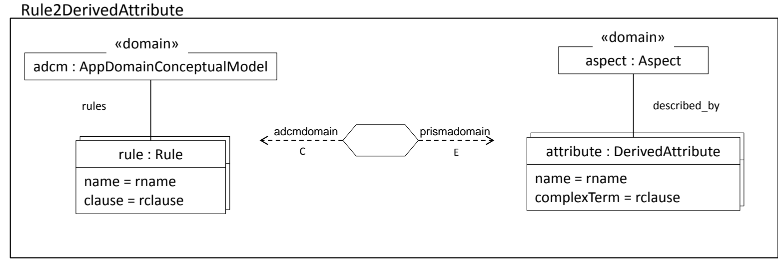


Figure 9.31: *Rule2DerivedAttribute* relation

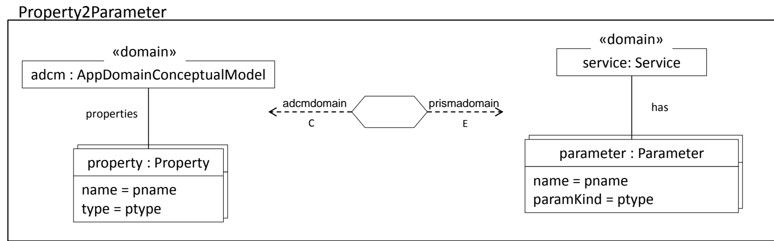


Figure 9.32: *Property2Parameter* relation

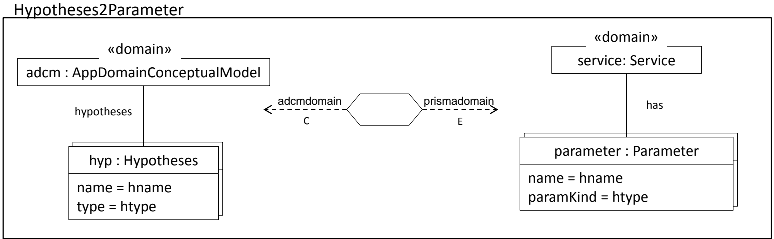


Figure 9.33: *Hypotheses2Parameter* relation

PROPERTY2PARAMETER RELATION Fig. 9.32 shows the *Property2Parameter* relation. It transforms the properties of the ADCM in the parameters of the services of a PRISMA aspect. The rule assigns the same names and types.

HYPOTHESES2PARAMETER RELATION The *Hypotheses2Parameter* relation (see Fig. 9.33) transforms the hypothesis of the ADCM in parameters of the services of an aspect. The relation assigns to them the same names and types.

CONNECTOR2CONNECTORPORTINTERFACE RELATION The *Connector2ConnectorPortInterface* relation (sowhn in Fig. 9.34) transforms each connector in the source domain to a PRISMA connector, and assigns to it the same name. The roles of the source connector are transformed to PRISMA ports (with their corresponding names) in the new connector. Moreover, the type of the port is also defined by creating the corresponding interface.

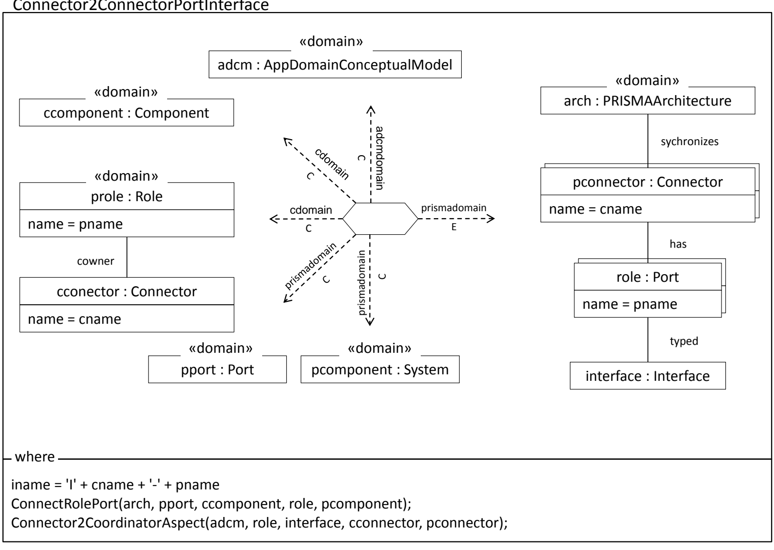


Figure 9.34: *Connector2ConnectorPortInterface* relation

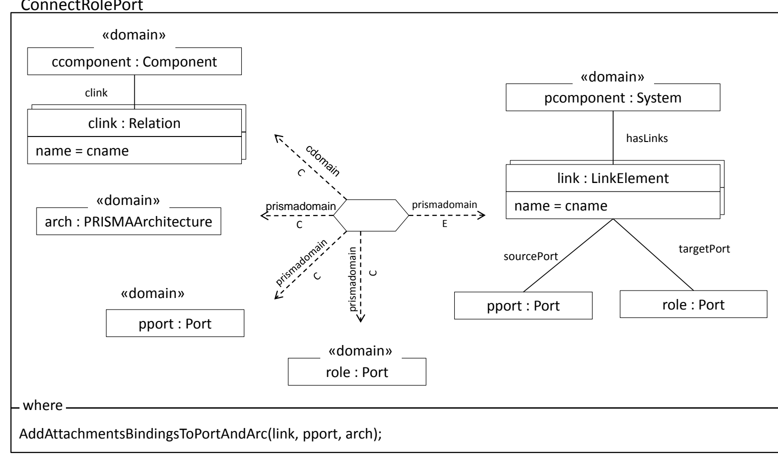


Figure 9.35: *ConnectRolePort* relation

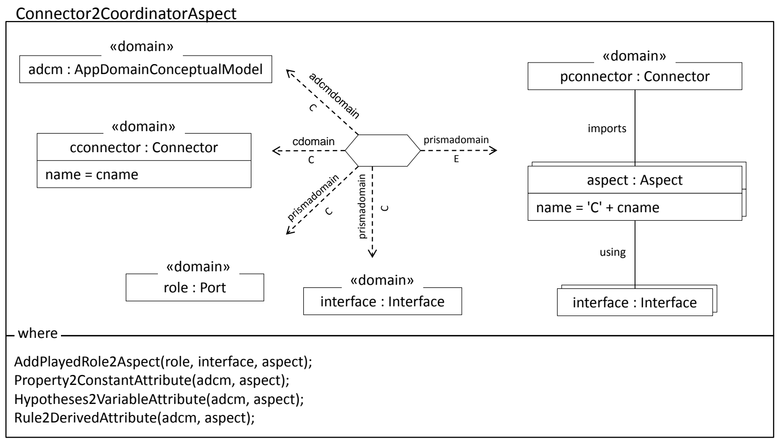


Figure 9.36: *Connector2CoordinatorAspect* relation

CONNECTROLEPORT RELATION Fig. 9.35 shows the *ConnectRolePort* relation. This rule transforms each relation which connects a role and a port in the component–connector metamodel to a LinkElement in the PRISMA metamodel. The created LinkElement attaches the corresponding ports between a PRISMA component and a PRISMA connector. The `where` clause invokes the *AddAttachmentsBindingsToPortAndArc* rule which adds the links that have

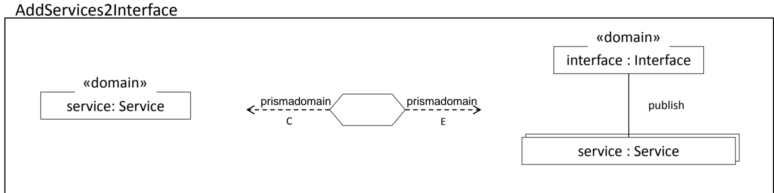


Figure 9.37: *AddServices2Interface* relation

been created to the root element of the architectural model (PRISMAArchitecture).

CONNECTOR2COORDINATORASPECT RELATION This rule, shown in Fig. 9.36, creates a coordinator aspect in every PRISMA component, and assigns to it the interfaces that it uses. The `where` clause invokes the *AddPlayedRole2Aspect*, *AddPlayedRole2Port*, *Property2ConstantAttribute*, *Hypotheses2VariableAttribute*, *Rule2DerivedAttribute* rules which have been explained before.

ADDSERVICES2INTERFACE RELATION As show in Fig. 9.37, the *AddServices2Interface* rule adds the services that have been created in the *PRISMA domain* to the interface that publishes them.

ADDPLAYEDROLE2ASPECT RELATION The *AddPlayedRole2-Aspect* is shown in Fig. 9.38. This rule adds the PlayedRole element to the corresponding PRISMA aspects, specifying the behaviour of an interface.

ADDPLAYEDROLE2PORT RELATION This simple rule, as shown in Fig. 9.39 assigns the PlayedRole to its corresponding port.

ADDATACHMENTSBINDINGSTOPORTANDARC RELATION The *AddAtachmentsBindingsToPortAndArc* rule (Fig. 9.40) adds the newly created attachments to the root element of the architectural model (the PRISMAArchitecture element). It also adds them to the related ports.

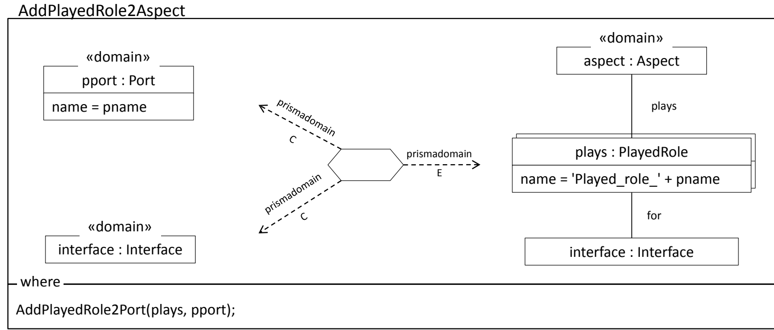


Figure 9.38: *AddPlayedRole2Aspect* relation

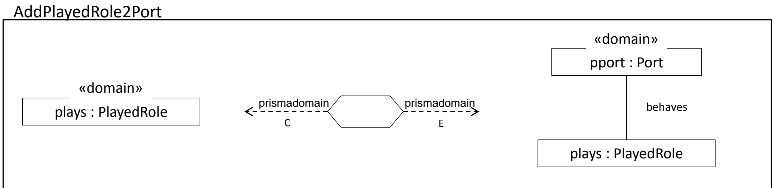


Figure 9.39: *AddPlayedRole2Port* relation

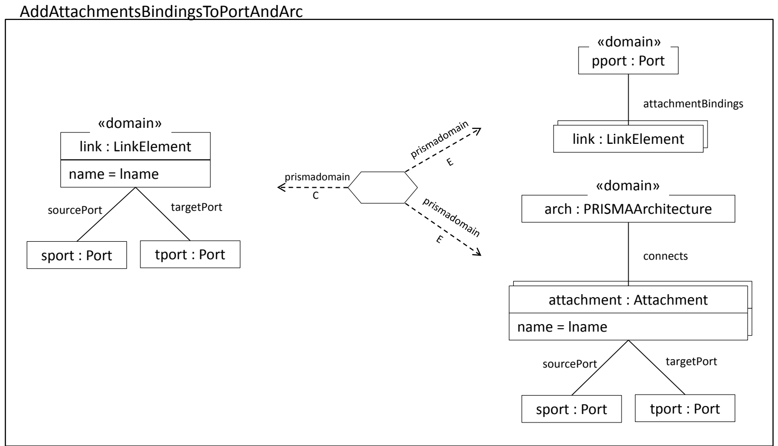


Figure 9.40: *AddAttachmentsBindingsToPortAndArc* relation

9.6 SUMMARY AND CONCLUSIONS

This chapter describes how to develop a SPL in a specific domain (Diagnostic Expert Systems in this thesis, but not limited to them)

by using Multi-Model Driven techniques. The development of such a kind of systems is a complex process because of elements that compose their architecture vary not only in their behaviour but also in their structure. This situation implies that several base architectures are obtained on the same reference architecture. Our approach uses QVT–Relations as the model transformations language to manage the variability along the whole process. Our approach also enhances the development of DES by applying SPL techniques, as they are useful when the members of a family of programs share a common design. This way, a specific design can be used in different products, reducing costs, time to market, effort and complexity. By applying MDA techniques, we are able to build systems that are platform-independent, and we can think about them from the problem perspective and not the solution perspective. This makes possible to apply such solutions to different domains. Moreover, we provide a framework with several technical spaces where modern software development languages and techniques coexist in a coordinated way (i. e., they conform a multi-model).

It is noteworthy to point out that the proposal covers the whole SPL life-cycle. First, we manage the variability for the first stage of the development process of DES. Such process continues until the base architecture is obtained. The second stage has been also implemented and the base architecture is then decorated with the application domain features. The result of the second stage produces a final and specific architecture. In our SPL the final architectural model is a PRISMA (Pérez Benedí 2006) model. As explained before, PRISMA is a framework to describe architectural models that provides the PRISMA-MODEL-COMPILER tool (Pérez et al. 2008). This tool is able to automatically generate executable C#.NET code, covering the whole development process.

Furthermore, in traditional approaches such as the BOM–Eager proposal, the group of base architectures is defined and implemented at design time of the SPL (domain engineering) and it remains unchanged throughout the whole life-cycle of the SPL (application domain). In the BOM–Lazy approach, the use of the *T1* transformation

allows us to move the creation of the base architectures to the application domain phase. This allows us to define the base architectures by using a set of rules that encode patterns of good design practices (as well as other design decisions), in a generic way. This avoids the need to define all of them explicitly. As the SPL grows in size, BOM–Lazy becomes an adequate approach to manage variability, as it supposes a great work to build *a priori* the base architectures for all the possible products of the SPL. Thus, the main effort is done in the domain engineering stage, where the acquired knowledge is formalized and encoded in a set of declarative rules (the knowledge is stored explicitly). So, it is not necessary to develop extensively all the possible combinations of base architectures (the knowledge is stored implicitly). That will increase the efficiency on the application engineering phase, where each base architecture is obtained only when it is needed by using the explicitly stored knowledge. Regarding to the *T2* transformation, the use of QVT–Relations raises the abstraction level in comparison with traditional FOP techniques, as we are dealing with high level concepts directly. In this sense, our proposal do not need to deal with XML documents o Extensible Stylesheet Language Transformations (XSLT) transformations, but it deals with equivalence patterns with describe the transformation step in a very natural way. Table 9.2 summarizes the domains, transformation rules and involved elements. In summary, we can conclude that the main characteristics of BOM–Lazy are:

1. Variability is managed at a high abstraction level (i. e. at the model level rather than at the program level).

2. The system variability is modeled using models that are separate from their functional models. The DSL for expressing the variability are suited for the domain, instead of adding tangled variability annotations directly to the functional models (UML or ADL) as other approaches have proposed.

3. Variability is operated by two orthogonal ways: one provided by the features of the domain, and another one provided by the features of the application domain.

4. The variability is given by instances of the conceptual models (DCM and ADCM).

5. Model generation and transformation are implemented using QVT–Relations in the Production Plan. In BOM–Lazy, the model transformations are resolved in an effective, scalable and user friendly way. Its expressiveness is also richer and clearer in comparison with traditional approaches which use Decision Tree and FOM/FOP techniques.

6. Various technological spaces are integrated, conforming a multimodel, to deal with the complexity of the problem. They are current trends in Software Engineering.

7. BOM–Lazy uses OMG standards and implements a generic approach to SPL development that can be applied to different domains, application domains, systems, and platforms.

8. BOM–Lazy offers an approach to build software applications in a simple way: the user only inputs the features of the domain and the application domain.

	RELATION NAME	V1 VARIABILITY MODEL	MODULAR MM	SKELETON MM
T1 TRANSF.	*ModulesModel2ComponentsModel*	—	ModulesModel	CCModel
	UseCase2Connector	UseCase	—	Connector
	Module2Component	Actor, UseCase, EntityViews/Reasoning	Module	Component
	Module2RolePort	UseCase	Module	Role, Port
	Function2Service	—	Function	Service
	Function2Relation	—	Function	Relation
	RELATION NAME	V2 VARIABILITY MODEL	SKELETON MM	PRISMA MM
T2 TRANSF.	*CCModel2PRISMAArchitecture*	—	CCModel	PRISMAArchitecture
	Component2Component	—	Component	Component
	Port2PortInterface	—	Port	Port, Interface
	Component2FunctionalAspect	—	Service	Aspect, Service
	Property2Parameter	Property	—	Parameter
	Hypotheses2Parameter	Hypotheses	—	Parameter
	Property2ConstantAttribute	Property	—	ConstantAttribute
	Hypotheses2VariableAttribute	Hypotheses	—	VariableAttribute
	Rule2DerivedAttribute	Rule	—	DerivedAttribute
	Connector2ConnectorPortInterface	—	Connector, Role	Connector, Port, Interface
	ConnectRolePort	—	Relation	LinkElement
	Connector2CoordinatorAspect	—	Connector	Aspect

Table 9.2: Rules and involved elements in the T1 and T2 transformations

10 AUTOMATED ANALYSIS OF FEATURE MODELS IN MULTIPLE: AN INDUSTRIAL EXPERIENCE

«Get the habit of analysis
–analysis will, in time,
enable synthesis to become your habit of mind»

— Frank Lloyd Wright
American architect, writer and educator, 1867–1959

Feature models are a suitable artifact to describe variability in product families. As such, the use of feature models to describe product lines is an increasing practice in industry today. To get the bigger profits, we have to deal with consistent and well-formed feature models.

However, as these models become larger, inconsistencies increase in number and complexity. It is essential to discover these errors and correct them in an easy and incremental way in order to avoid incorrect product designs and extra costs derived from a late detection.

Feature models must be reliable, since they are used as the input in many other SPLE processes such as code generation (Czarnecki and Eisenecker 2000; Czarnecki and Antkiewicz 2005) or feature oriented MDD (Batory 2003; Trujillo 2007; Cabello and Ramos 2009; Cabello et al. 2009). The analysis of feature models is an important

task, and it must be done before starting any other activity to avoid the propagation of errors. If we are dealing with large-scale feature models it is almost impossible to perform the analysis manually and we need a tool allowing us to do it in an automated way. Nowadays, there are different proposals to automate the analysis of feature models (Benavides et al. 2010). One of the most interesting tools is FeAture Model Analyser (FAMA) (ISA 2011a). This framework, developed by researchers of the *Universidad de Sevilla*, provides a way to perform analysis operation over feature models. Its main advantage lies in its formal semantics, which avoids misinterpretation. As it can use either constraint programming (Tsang 1995), boolean satisfiability techniques (Cook and Mitchell 1997) or binary decision diagrams (Bryant 1986) to represent feature models rigorously, it becomes a reliable tool to automate feature model analysis.

The *Ingeniería del Software y Sistemas de Información (ISSI)* Research Group, where this thesis has been developed, has started a collaboration with *Rolls-Royce plc* in the context of the "MULTIPLE: Multimodeling Approach For Quality-Aware Software Product Lines" project, funded by the Spanish Ministry of Science and Innovation (ref. TIN2009-13838). Thanks to this collaboration, we have been able to study the variability which arises when developing software for embedded systems in the aero-engine industry.

The main purpose of this chapter is to provide a discussion about feature modeling in industry by means of performing automated analysis over a real industrial feature model and to show the results obtained after its complete analysis. The analysis is done using the MULTIPLE framework, presented in chapter 8, which relies on the FAMA analysis tool.

10.1 CONTEXT AND MOTIVATION

Nowadays, performing a complete analysis of a large-scale feature model represents a big challenge. In this context, having a set of tools to ease this task is a must. It is also a must to keep on feeding the discussion about what kind of errors are more common in large-

scale industrial feature models, and how to correct them. We need, then, to get relevant results obtained from the analysis over real large-scale industrial feature models.

There exist different automated analysis tools which are based on a specific representation (model) of a feature model. This way, our model must be adapted to fit a specific notation in order to be analysed. To analyse our feature model by means of an automated analysis tool is not an immediate task. Probably our model wouldn't adapt totally to the representation accepted by the tool. In addition, if our model is too large, it is possible that it contains syntactical errors. These errors are hard to detect manually and can that make the model not analysable.

Feature models must be specially reliable when they are used in MMDPLE processes. for this reason, we need automatic tools to analyse large-scale feature models as they are more error-prone.

Thus, to analyse our model, we need one or more intermediate steps in which:

1. The syntactic correctness of the model is checked.
2. The model is adapted to the notation used by the automated analysis tool.

In chapter 8 we presented the MULTIPLE framework. This tool incorporates feature modelling capabilities in a MDA environment, implementing a set of components to manage extended feature models. Next, we present a case study and we present how we have used MULTIPLE to represent and analyse a feature model:

1. A parser processes data from the source model and creates XMI instances that satisfies the MULTIPLE feature metamodel. Moreover, the syntactic analysis and correction of the model is automated.
2. A model transformation (*MultipleFeatures2FamaFeatures*, see Appendix D) is used to generate a feature model understandable by FAMA.
3. The integrated component to validate FAMA feature models is used to analyse the large-scale feature model providing:

a) Error detection.
b) Error classification.
c) Solutions to the errors.
d) Model correction rates.

10.2 CASE STUDY

To illustrate the capabilities of the MULTIPLE *framework we have analysed an industrial and large-scale feature model from an aircraft engine manufacturer.*

Aircraft engines are big and complex systems which are made up by several components. For example, a jet engine must have specific subsystems to guide the air and fuel flow, lubrication, water injection, noise levels, etc. To control these elements several devices (such as turbines, compressors, valves, actuators, gearboxes…) must be controlled. To keep these elements in working order different electronic controllers are used. Such controllers must be programmed and coordinated, which is a quite complex task.

Three are the most important aircraft engine companies worldwide: *GE Aviation* (GE 2011), subsidiary of *General Electric*); *Rolls-Royce plc* (Rolls-Royce 2011); and *Pratt & Whitney* (Pratt & Whitney 2011), subsidiary of United Technologies Corporation (UTC). Such companies build different engine models, which are installed in different aircrafts. In this situation it is very important to cope with the variability problem to ease the development of new engine variants

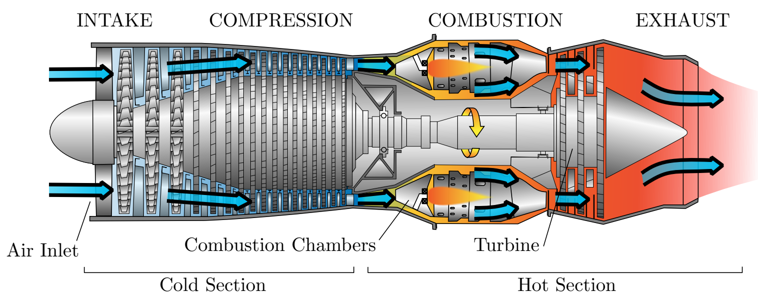


Figure 10.1: Basic components of a gas turbine engine (FAA 2007)

which can be sold to different aircraft manufacturers, such as Boeing (Boeing 2011) or Airbus (Airbus 2011).

10.2.1 *Feature modeling in Rolls-Royce plc*

An aero-engine (Fig. 10.1) can be seen as a complex operating unit that works thanks to the interaction of different mechanisms (turbine, compressor, combustion chamber...). These mechanisms can be of very different types and can be configured in different ways. There are lots of details to take into account in the development of those artifacts, as well as a high degree of variability. As a consequence, Rolls-Royce has characterized the variability of aircraft engine by using feature models. But, developing feature models is a hard task as they are very complex and contain much variability. Moreover, the analysis of these models (which is necessary to assure that they are correct) is mandatory.

The initial feature model has been developed using the FeatureRSEB *proposal, which is a subset of the* MULTIPLE *proposal for feature modeling.*

The feature model developed by *Rolls-Royce plc* uses the PLUSS approach, which is, basically, the original FODA proposal adding the OR group. PLUSS provides *mandatory* or *common* features, *optional* features, *single adaptor* features and *multiple adaptor* features that may have cross-tree *requires* and *excludes* relationships with other features. Since PLUSS is almost equivalent to FODA, it can

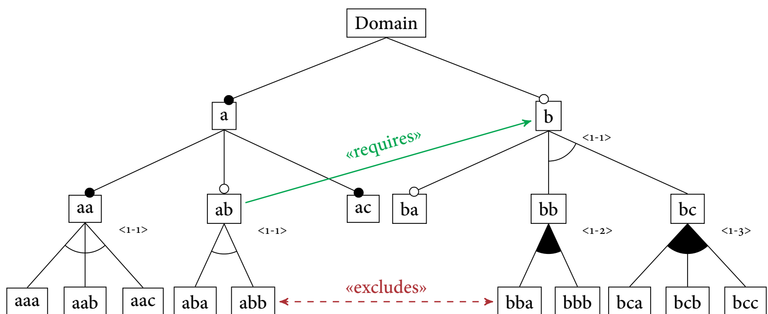


Figure 10.2: Cardinality-based feature model equivalent to the one shown in Fig. 4.6

PLUSS	CARDINALITY-BASED
Common	Mandatory ([1..1])
Optional	Optional ([0..1])
Single adaptor	XOR (<⟨1-1⟩)
Multiple adaptor	OR (⟨1-k⟩)
Requires	Implies
Prohibits	Excludes

Table 10.1: Correspondences between Cardinality-based feature models and PLUSS

be implemented in a subset of the cardinality-based proposal of MULTIPLE. That way, it is possible to transform a PLUSS feature model to a cardinality-based one. Fig. 10.2 shows a cardinality-based feature model which is equivalent to the feature model shown in Fig. 4.6 (page 55). Notice that the mappings between both models are straightforward. Table 10.1 summarizes the correspondences between the PLUSS models and cardinality-based feature models[1].

10.2.2 *Analysis process overview*

We have shown that transforming a PLUSS feature to a cardinality-based one is straightforward. This allowed us to import the feature model developed by Rolls-Royce plc into the MULTIPLE framework, with the aim of analysing its correctness. This way, we can test the

1 Some relationships that appear in our cardinality-based proposal are not shown for the sake of clarity. The *biconditional* relationship can be expressed by using two different requires relationships in the PLUSS proposal. However, the *use* relationship that we propose does not have a correspondence, as PLUSS does not support cloning of features.

scalability of the MULTIPLE framework when dealing with real and large-scale feature models provided by the industry. We used FAMA to implement a method that allows us to perform a complete analysis (syntactic and semantic) of the Rollws-Royce feature model. Our proposal is outlined in Fig. 10.3. As our metamodel is a superset of the PLUSS metamodel, we can represent any PLUSS feature model an get the benefits that the MULTIPLE framework provides. This way, the imported feature model can be enriched with complex model constraints written in FMCL; it can be edited using the MULTIPLE feature modeling editor; or we can use MULTIPLE to create model configurations and check whether they are valid or not.

For the analysis of the industrial feature model we propose to use the MULTIPLE *feature model as the pivot artifact, using* EMF *for its canonical representation.*

10.2.2.1 *Process details*

The tool support for the PLUSS approach is built on top of the Dynamic Object-Oriented Requirements System (DOORS) tool (IBM 2011). To work with the feature model in MULTIPLE, we must first export the feature model as a Comma-Separated Values (CSV) file. The analysis process is divided in two parts. First (1), an analyzable

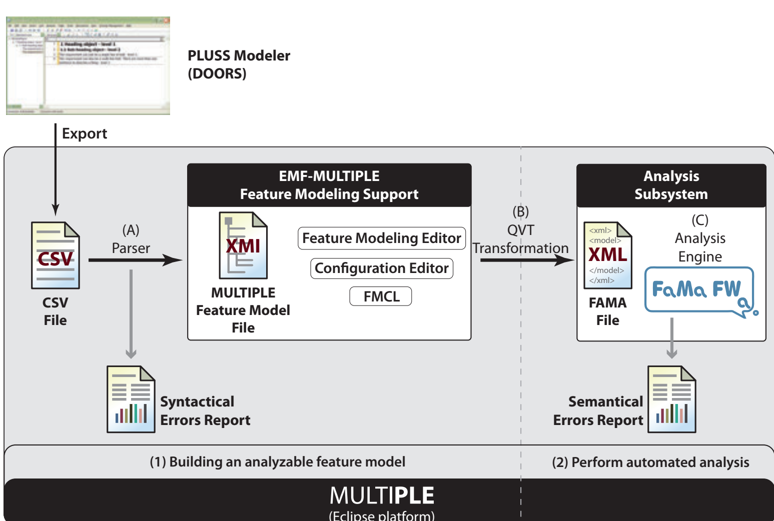


Figure 10.3: Schema of the proposal

MODULE NAME	MODULE NUMBER	ABSOLUTE NUMBER	PARENT NUMBER	OBJECT NUMBER	OBJECT HEADING	FEATURE NODE	VARIATION	REQUIRES	PROHIBITS
Anon. Module 1	0000706b	178	166	1.1.7	Anon. Feature 1	Feature	Optional		
Anon. Module 1	0000706b	188	166	1.1.8	Anon. Feature 2	Feature	Optional	233,128[b]	
Anon. Module 1	0000706b	183	166	1.1.9	Anon. Feature 3	Feature	Common		500[c]
Anon. Module 1	0000706b	135	364	1.2	Anon. Feature 4	Feature Group	Common		
Anon. Module 1	0000706b	392	135	1.2.1	Anon. Feature 5	Feature Group	Common		
Anon. Module 1	0000706b	393	392	1.2.1.1	Anon. Feature 6	Feature	Common		
Anon. Module 1	0000706b	394	392	1.2.1.2	Anon. Feature 7	Feature Group	Optional		

[a] Module and feature names have been anonymized.

[b] Actual contents of this field refer to the fully qualified name of the referenced features, i. e.: ‘‘Anon. Feature 8,0000706b,233,5.2.2.2|Anonymized Feature 9,0000706b,128,2.1.3’’.

[c] Actual contents of this field are ‘‘Anon. Feature 10,0000706b,500,3.2’’.

Table 10.2: Source model extract

feature model must be built. For that purpose, the *Parser* component (Fig. 10.3, element A) maps the source feature model (contained in the CSV file) that uses PLUSS notation into a XMI instance of our metamodel.

Building an analyzable feature model also implies that a syntactic analysis is performed to detect and fix possible errors in the source model. The syntactic errors are here introduced by the users who created the feature model manually. These errors are produced because DOORS is a requirements management tool, and it is not specially designed for feature modeling, so no checking capabilities are provided.

Once the source feature model has been translated to an equivalent feature model in MULTIPLE, we can use any of the framework tools, including model transformations. This way, it is very simple to transform a feature model to any other model type. Using a QVT model transformation—Fig. 10.3, (B)—the feature model is projected to the FAMA representation.

The second part of the process (2) comprises the analysis using FAMA—semantic analysis, element (C) in Fig. 10.3. Both parts of the process are performed by the prototype in a way transparent to the user.

10.2.3 *Source Model structure*

The source 10.2 feature model contains 1195 features, so we consider it a large-scale feature model. Table 10.2 shows an small fragment of the model that illustrates its actual structure as a CSV file. Module and feature names have been removed. We must remark that the data used in this case study are protected by the non-disclosure agreement signed between *Roll-Royce plc* and the ISSI research group. As a consequence, tables, figures and examples in the remaining of this chapter will be anonymized to obey this agreement.

The source feature model has been extracted as a plain text file which uses a comma-separated value structure.

As table shows, the model is codified as a plain text in a tabular structure, in which features are defined as table rows. An screenshot

which shows the actual contents of the file is shown in Fig. 10.4. The meaning of each columns of the table is detailed below:

Module Name — Contains the name of the SPL module to which the feature belongs to.

Module Number — Contains the number of the SPL module to which feature belongs to.

Absolute Number — Contains the number that identifies the feature in the module.

Parent Number — Contains the number of the parent feature.

Object Number — Contains the number that represents the feature in the feature hierarchy.

Object Heading — Contains the feature name.

Feature Node — It may contain two different values: ''Feature'' or ''Feature Group'', that indicate if the feature has children or not (leaf).

Variation — Contains the type of variation that represents the feature. It can have four different values: ''Common'', ''Optional'', ''Single'' or ''Multiple'', according to the PLUSS notation.

Requires — Contains the feature or features that this feature requires when it is selected (should be also selected).

Prohibits — Contains the feature or features that can not be selected when this feature is selected.

10.2.4 *A step by step description of the process*

This subsection shows visually the steps that have been described before, and shows how they have been put in practice using the MULTIPLE framework.

Figure 10.4: Rolls-Royce PLUSS feature model

10.2.4.1 *Step 1: the initial model file*

First, we need to include in the workspace the feature model that we want to analyze (i. e., the Rolls-Royce feature model). This file is called *RollsRoyceSample.csv* (Fig. 10.4).

10.2.4.2 *Step 2: Parser execution and syntactical analysis*

Second, we execute the parser, which will create a new instance of the pivot model (i. e., a MULTIPLE cardinality-based feature model). To execute the parser, the user must use the contextual menu as shown in Fig. 10.5. The result model, called *RollsRoyceSample.features*, is equivalent to the original model. In Fig. 10.6 a simplified version of this file is shown (to see different relationship types) in the MULTIPLE feature modeling tree editor.

Additionally, a console with the results of the syntactical analysis is shown (see Fig. 10.7). There, the model errors found and the solutions applied to solve them are listed. When an error is detected, the parser displays the following information: line number were the error was found, the cause of the error and the solution adopted.

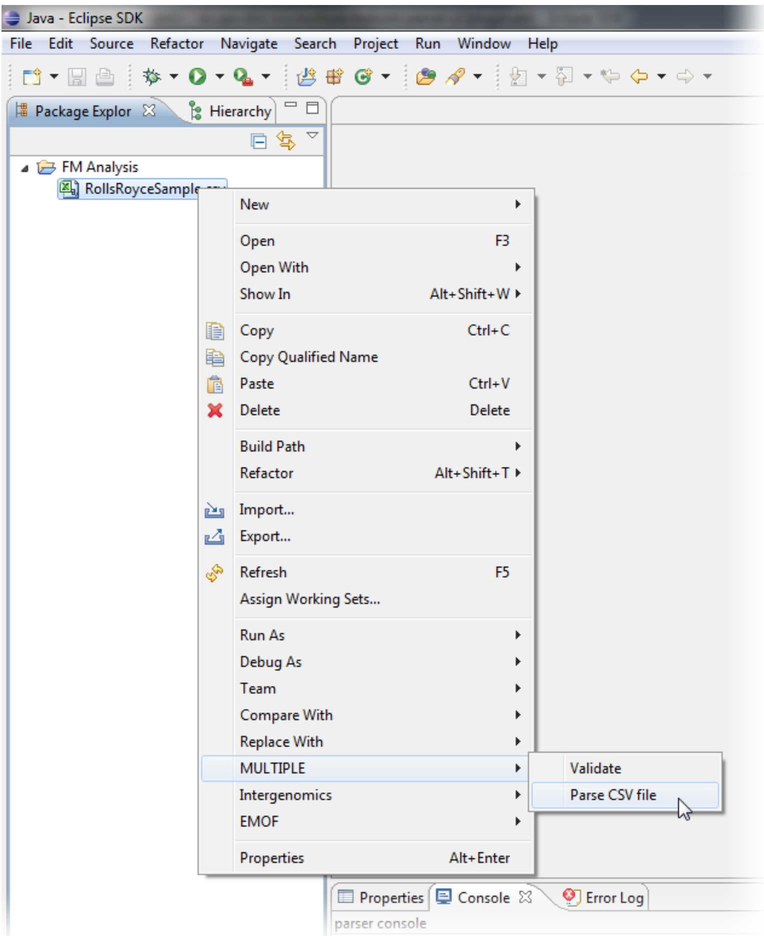


Figure 10.5: Running the PLUSS parser

It is worth mentioning that the produced feture model can be opened using the graphical feature model editor (shown in Fig. 8.12, page 138). However, due to the enormous size of the Rolls-Royce feature model this graphical representation it is not the best option to visualize and navigate the model.

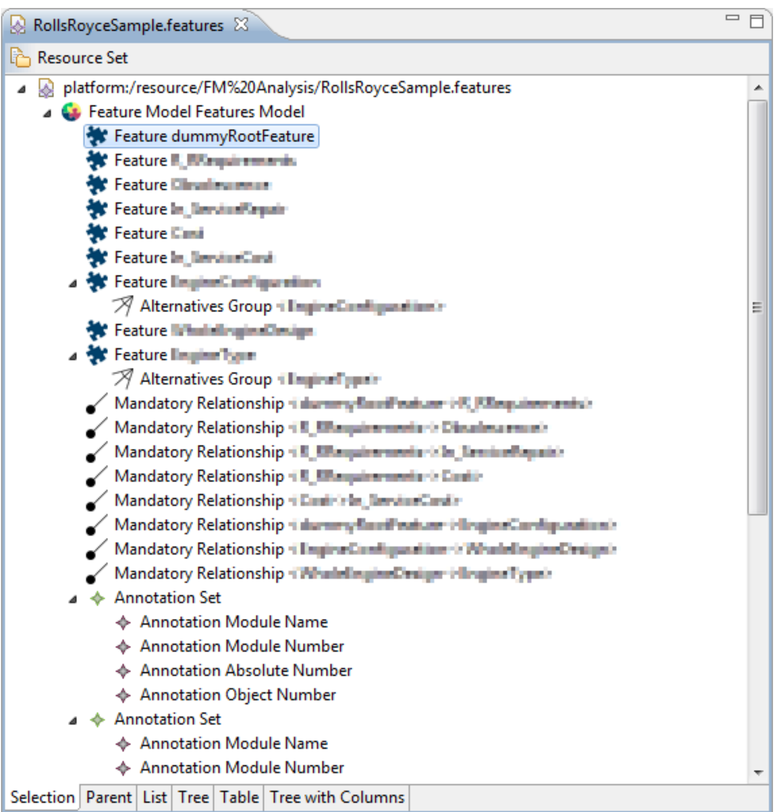


Figure 10.6: The Rolls-Royce feature model shown in the MULTIPLE feature modeling tree editor

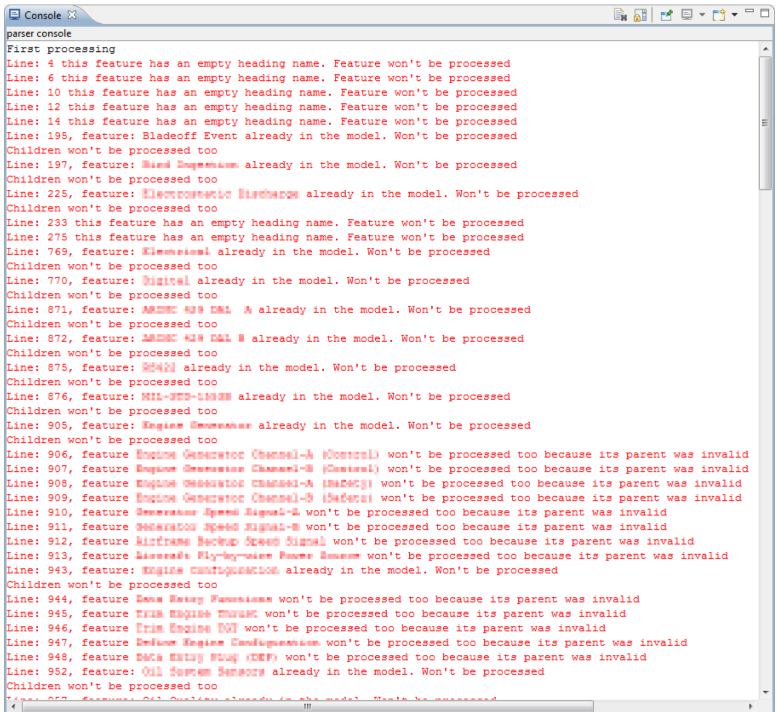


Figure 10.7: Console showing the syntactical analysis results

10.2.4.3 *Step 3: Model transformation execution*

Third, we must execute the *MultipleFeatures2FamaFeatures QVT* transformation. which will generate an *XML* file taht can be loaded into the FAMA framework. The transformation is executed in the usual way as Fig. 10.8 shows.

As a result, a file called *result.fama* is generated. This file can be open by using a tree editor (see Fig. 10.9) to easily explore its contents. This is possible thanks to the FAMA metamodel support provided by MULTIPLE. It must be pointed out that the persistence format that this editor uses is not the standard XMI serialization used by EMF, rather it is the native XML format supported by the FAMA framework as Fig. 10.10 shows.

A traces model is also obtained. This model stores the mappings between the source model and the target model, which are useful to trace errors back when they are found by the FAMA framework.

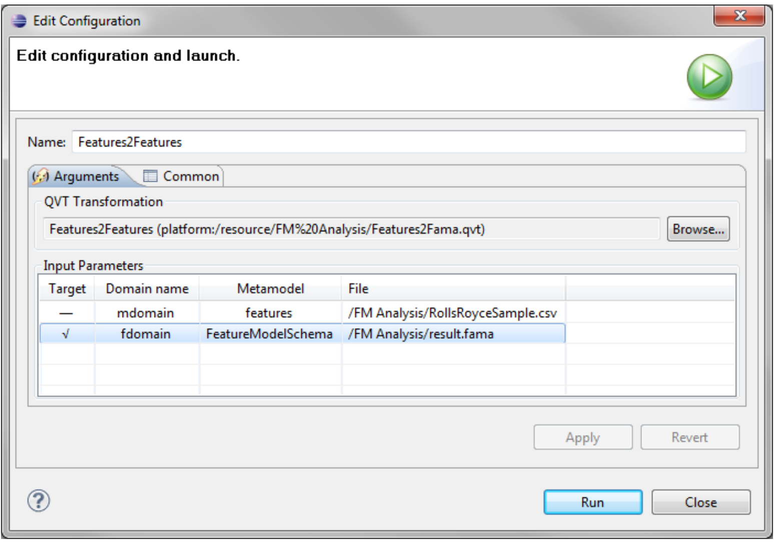


Figure 10.8: Dialog box showing the the *MultipleFeatures2FamaFeatures* transformation is ready to be executed

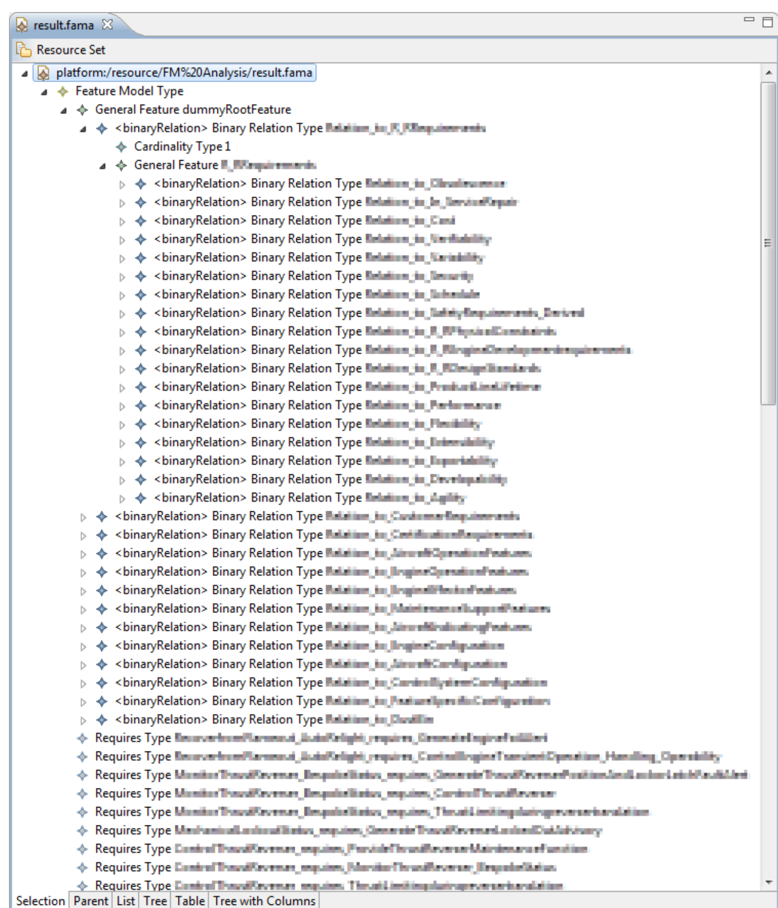


Figure 10.9: Rolls-Royce model represented as a FAMA model in the FAMA tree editor

result.fama

```xml
<?xml version="1.0" encoding="ASCII"?>
<feature-model
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://www.tdg-seville.info/benavides/featuremodelling/feature-model.xsd">
  <feature name="dummyRootFeature">
    <binaryRelation name="[illegible]">
      <cardinality max="1" min="1"/>
      <solitaryFeature name="[illegible]">
        <binaryRelation name="[illegible]">
          <cardinality max="1" min="1"/>
          <solitaryFeature name="[illegible]"/>
        </binaryRelation>
        <binaryRelation name="[illegible]">
          <cardinality max="1" min="1"/>
          <solitaryFeature name="[illegible]"/>
        </binaryRelation>
        <binaryRelation name="[illegible]">
          <cardinality max="1" min="1"/>
          <solitaryFeature name="[illegible]">
            <binaryRelation name="[illegible]">
              <cardinality max="1" min="1"/>
              <solitaryFeature name="[illegible]">
                <binaryRelation name="[illegible]">
                  <cardinality max="1" min="1"/>
                  <solitaryFeature name="[illegible]"/>
                </binaryRelation>
                <binaryRelation name="[illegible]">
                  <cardinality max="1" min="1"/>
                  <solitaryFeature name="[illegible]"/>
                </binaryRelation>
              </solitaryFeature>
            </binaryRelation>
            <binaryRelation name="[illegible]">
              <cardinality max="1" min="1"/>
              <solitaryFeature name="[illegible]"/>
            </binaryRelation>
            <binaryRelation name="[illegible]">
              <cardinality max="1" min="1"/>
              <solitaryFeature name="[illegible]">
                <binaryRelation name="[illegible]">
                  <cardinality max="1" min="1"/>
                  <solitaryFeature name="[illegible]"/>
                </binaryRelation>
              </solitaryFeature>
            </binaryRelation>
            <binaryRelation name="[illegible]">
              <cardinality max="1" min="1"/>
              <solitaryFeature name="[illegible]"/>
            </binaryRelation>
```

Figure 10.10: Rolls-Royce model represented using the FAMA XML serialization format

10.2.4.4 *Step 4: Running the FAMA analysis engine*

MULTIPLE provides a contextual menu to perform the FAMA analysis operations. Fig. 10.11 shows the operations which are available: *Detect and explain errors*, *Number of products*, *Products* and *Variability degree*. The results will by displayed in the *Fama Analysis Console* as shown in Figs. 10.12, 10.14 and 10.13 respectively. Only a small

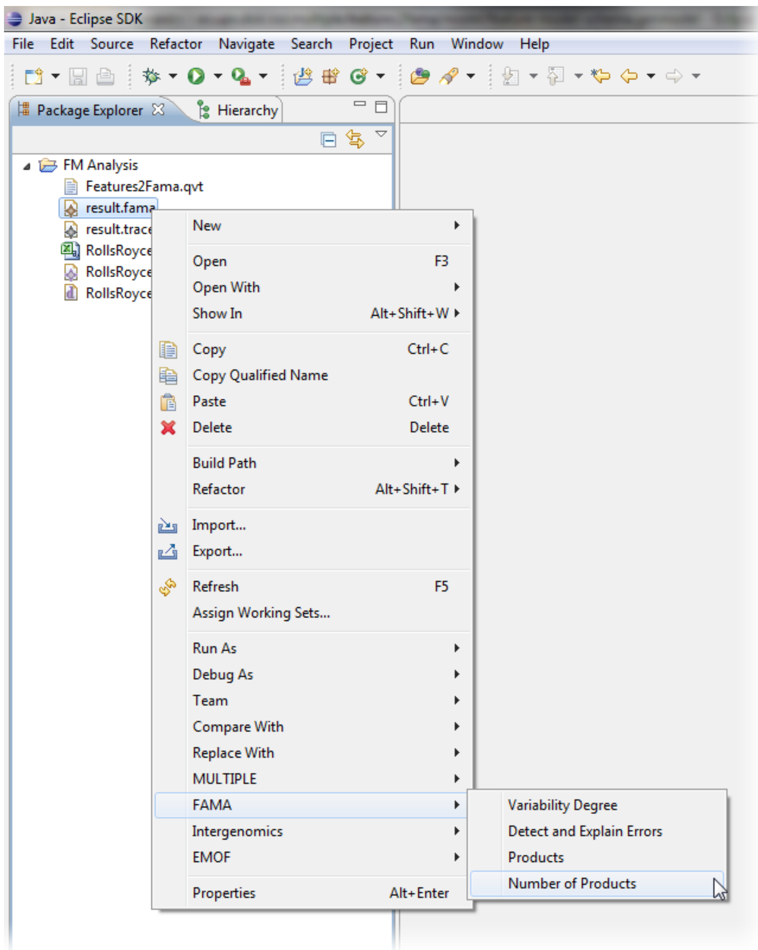


Figure 10.11: Running FAMA analysis from the MULTIPLE user interface

number of operations have been finally included, as the others did not provide results in time (see section 10.3.4). This way, only the working operations have been included in the prototype.

```
Console
Fama Analysis Console
WELCOME TO THE FAMA ANALYSIS CONSOLE. It might take some time to compute the result...

File result.fama loaded succesfully

The number of products is: 3073
```

Figure 10.12: Calculating the number of products using FAMA

```
Console
Fama Analysis Console
WELCOME TO THE FAMA ANALYSIS CONSOLE. It might take some time to compute the result...

File result.fama loaded succesfully

5 error(s) found:

1: False-mandatory Feature: [illegible]

2: False-mandatory Feature:
[illegible]

3: False-mandatory Feature: [illegible]

4: Dead Feature: [illegible]

5: Dead Feature: [illegible]

Timeout looking for explanations!
```

Figure 10.13: Detecting errors using FAMA

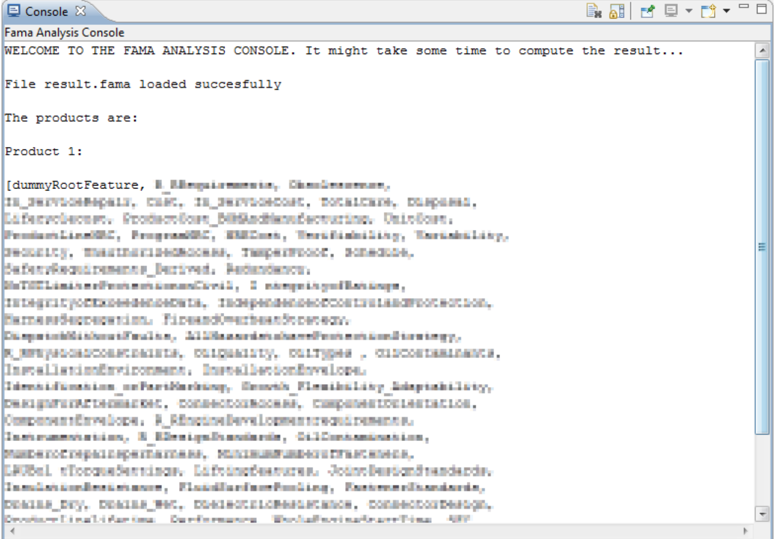


Figure 10.14: Calculating products using FAMA

10.3 INTERPRETING THE OBTAINED RESULTS

This section contains the explanation of the errors found during the analysis process. We also classify and provide solutions to those errors.

10.3.1 *Syntactic analysis*

During the process to adapt the PLUSS model to our notation, we have found some errors that make the source feature model inconsistent. These errors can be classified as follows:

Before using FAMA to validate the feature model, a pre-process must be carried out to tidy up the source model, i. e., any syntactical error in the model must be found and corrected.

EMPTY FEATURES We identified that the model contains unnamed features (Fig. 10.15a). These *empty* features are defined as mandatory, so they must be present in every product of the Software Product Line.

Every feature in the model has to define a functionality, and these *empty* features do not contribute in any way to the SPL.

Solution — We ignore empty features, since they are not significant. Moreover, the console displays the line of the model in which the feature is placed and the solution taken (Fig. 10.15b).

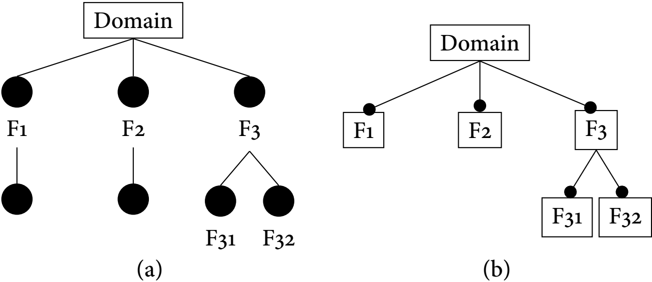


Figure 10.15: Empty features in the original feature model 10.15a and a possible solution 10.15b

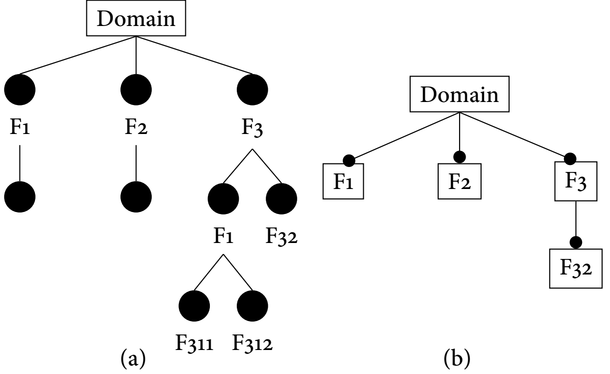


Figure 10.16: Duplicated features in the original feature model 10.16a and a possible solution 10.16b

DUPLICATED FEATURES The feature model contains duplicated features. This means that we have two or more features placed in different hierarchical positions due to the kind of feature model that we are working with.

One of the main goals of defining a feature model is to capture variabilities and commonalities between the different products. This way, the feature model reflects every possible feature combination of the products of an SPL.

Every feature has to be unique. In cardinality-based feature models we can have cloned features, but we can not have duplicate features inside the same model in different hierarchy locations.

Solution — By default, we leave just the first appearance of the feature. If a feature to be removed contains children, they will be removed too (Fig 10.16b). The removed features could represent different product characteristics that have been incorrectly defined using the same name. These errors also lead us to the problem of removing some valid features obtaining less potential products. Thus, the user is warned about the error and its location to be revised.

MEANINGFUL USE OF FIELDS As we commented at the beginning of this chapter, the source feature model is coded in plain text. In the source model we find the *Feature Node* field, which informs whether the feature belongs to a group (*Single* or *Multiple*) or not. This field can contain two values: *Feature Group* and *Feature*. We can find features defined as groups that sometimes have children and sometimes are *leaf* features. The same happens when they are defined as *Features*. Moreover, this field is often undefined and even it occasionally seems dependent on the variability. There is no pattern in the use of this value.

Solution — We did not rely on this *Feature Node* field to identify the parent-child relationships between features. We deduce them from other fields, such as the variability and the parent number.

The main problem arises while defining features in the model. There is no pattern followed to set a feature to be a *Feature Group* or a *Feature*. We take as correct practice to define a feature as a *Feature Group* if it has children and as a single *Feature* otherwise.

Table 10.3 represents a fragment of the PLUSS feature model. *Declared Feature Node* column contains the original values of this field in the source model and *Expected Feature Node* column contains the correct values expected given the tree structure. Features with the *Feature Node* field value in red are some examples of the errors found. As we can observe, feature *F2* is defined as a Feature, but it has two children. However, *F21* is defined as a group and it does not have children. The same happens with feature *F12*, it has children but it is not defined as a group.

Fig. 10.17 shows the correct representation in Cardinality-based notation.

ID	DECLARED FEATURE NODE VALUE	EXPECTED FEATURE NODE VALUE
Domain	Feature Group	Feature Group
F1	Feature Group	Feature Group
F11	Feature	Feature
F12	Feature	Feature Group
F121	*Empty*	Feature
F122	*Empty*	Feature
F123	*Empty*	Feature
F2	Feature	Feature Group
F21	Feature Group	Feature
F22	Feature	Feature

Table 10.3: Example of features incorrectly defined in the source CSV file

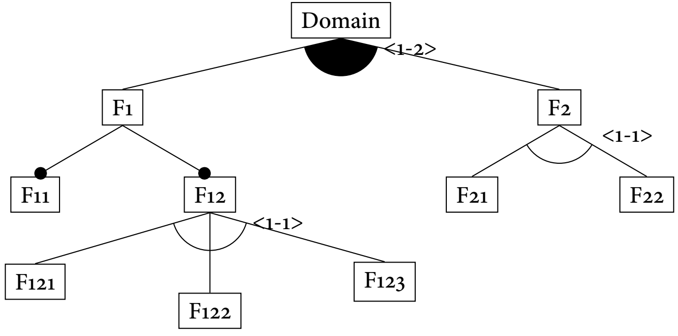


Figure 10.17: Correct representation using cardinality-based notation

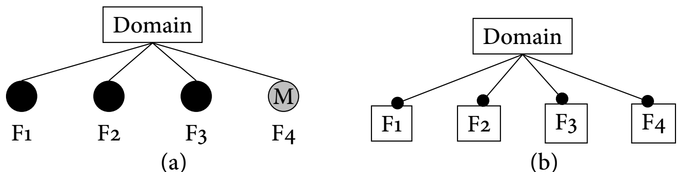


Figure 10.18: Example of ambiguous use of feature groups 10.18a and a possible solution 10.18b

AMBIGUOUS USE OF VARIABILITY *Multiple Adaptor* features represent an *at-least-one-out-of-many* selection that has to be made among a set of features.

In the model we can find the use of this *Multiple* variability having a set of just one feature (Fig. 10.18a), so there is no real option. You cannot choose between more than one option because you have a multiple Group with just one child. So we have an incorrect use of the *Multiple* variability.

Solution — In the cases where a feature valued as *Multiple* is part of a group of just one child, we replace this relationship with a *mandatory* relationship (Fig. 10.18b). Since the meaning of the *Multiple Adaptor* is *at-least-one-out-of-many*, *mandatory* is the most accurate relationship to represent it.

10.3.2 *Semantic Analysis*

As we have a large-scale feature model, we need an automated way to analyse the information inside. In MULTIPLE we use FAMA to do this.

The semantic analysis is performed by using FAMA, and finds any inconsistency which make the model totally or partially invalid.

In order to obtain significant results, we have selected only those operations which apply to feature models (i. e. model-checking operations), and not feature model configurations since our framework already provides this functionality.

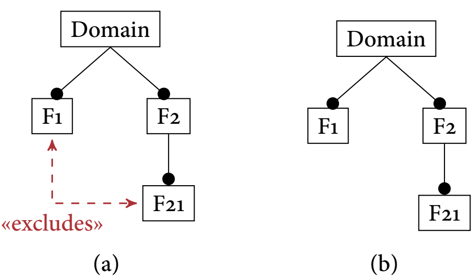


Figure 10.19: Invalid relationship 10.19a and a possible solution 10.19b

VALIDATION The result of the validation is that the model is unsatisfiable. The PLUSS Feature model is invalid because it contains two cross-tree relationships of type *Excludes* that are not well defined since they exclude *mandatory* features (Fig. 10.19a).

When we define *mandatory* features, they cannot be excluded by another mandatory feature (if all the parents of both features are also mandatory) because that fact contradicts their mandatory definition.

Solution — When this kind of contradictions arise, there are two ways to proceed:

1. To remove the troubled relationship (Fig. 10.19b).
2. To check the definition of features in the model to identify if those features should be *optional* instead of *mandatory*.

We took the first solution (remove troubled relationships) because it is less intrusive with the structure of the SPL. The tool also warns the expert user to allow him/her to check the model and fix whatever is needed.

PRODUCTS & NUMBER OF PRODUCTS This operation calculates all the possible configurations that the feature model represents. The analysis of the feature model detected 3073 different potential products in the product line. It is a high number because we are

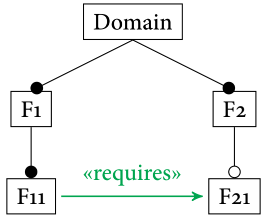


Figure 10.20: Example of a false-mandatory feature

dealing with a large-scale feature model. The *products* operation also provides the features combination for every product in the console.

ERROR DETECTION Besides validation errors, error detection results revealed five additional errors in the model:

1. *Three* false-mandatory features

   False-mandatory features are those features that behave as *mandatory* but they are not defined as *mandatory*.

   For instence, feature *F21* in Fig. 10.20 is a false-mandatory feature because it is defined as *optional* but it is going to be present in every product of the SPL. This is because of the *Requires* relationship between the *mandatory F11* feature and *F21*.

   Solution – To change the type of the feature to *mandatory*.

2. *Two* dead-features

   The most frequent reason for having dead features (features that are present in no products) is the existence of contradictory relationships. Fig 10.21a shows an example of dead features. Features under the *Domain* feature are part of a *XOR* group (or *Single Adaptor*, i. e., exactly-one-out-of-many). Feature *F1* requires feature *F2* and vice versa. In a XOR group we can have just one feature selected, so this two *Requires* relationships contradict the group definition.

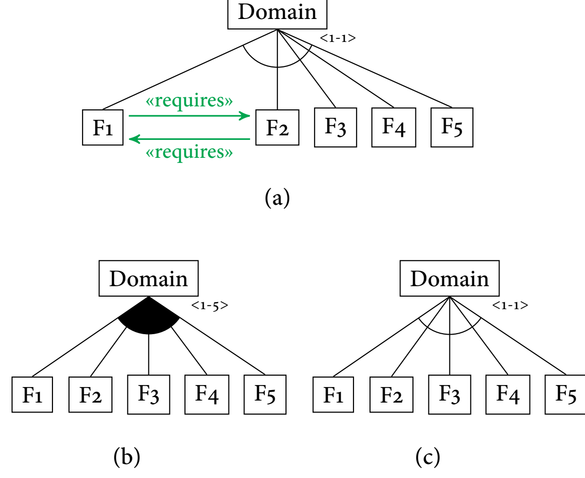


Figure 10.21: Example of dead features (features F1 and F2 in 10.21a) and two possible solutions (10.21b and 10.21c)

Solution — In this case, the problem can be at the *Group* definition. An *OR Group* or *Multiple Adaptor* (at least-one-out-of-many) would solve the problem (Fig. 10.21b), because it allows us to select more than one feature.

Another solution is to remove the *Requires* relationships (Fig. 10.21c).

The solution would vary depending on what the model aims to represent.

10.3.3 *Conclusions about the analysis results*

Table 10.4 shows the results of the analysis performed that give us the necessary data to determine the correction of the feature model. This table reflects the feature model percentage of correction grouped by element type.

The *Source Model* column refers to the elements of the original feature model, and the *Target Model* column refers to the elements

	SOURCE MODEL	TARGET MODEL	% CORRECT
Number of Features	1195	1016	85.02
Optional Features	295	263[a]	89.15
Mandatory features	833	692	83.07
Single features	53	53	100
Multiple features	14	5	35.71
Number of groups	258	19	7.36
Implies	70	63[b]	90
Excludes	6	4[c]	66.67
Total elements	1529	1099	71.88

[a] 266-3 false mandatory

[b] 65-2 relationships which cause dead features

[c] 6-2 relationships that make the model void

Table 10.4: Percentages of correction grouped by element type

of the corrected model obtained after the syntactic and semantic analyses. The percentage of correction has been calculated as follows:

$$Correction\ (\%) = \frac{Correct\ model\ elements}{Total\ (original)\ model\ elements} \times 100$$

According to the results obtained, we can order the number of errors as follows:

Groups defined > Multiple features >
Excludes relationships > Mandatory features >
Optional features > Implies relationships >
Single features

10.3.4 *Efficiency and limitations of the automated analysis tool*

This section describes the limitations found while performing some of the operations that the FAMA Framework provides. Most of these limitations are derived from the scale of the feature model we were working with (1016 features). We used the FAMA 1.1.1b version released on April 14th, 2011, and a standard desktop computer[2].

The efficiency and problems found while performing each analysis operation are listed below:

10.3.4.1 *Validation*

This operation works well with our model, providing a quick result in about 3 sec. However, problems arise when the model is void and the tool tries to look for error explanations. The tool is unable to give a response in an acceptable time.

10.3.4.2 *Products and Number of Products*

These operations also work reasonably well with the model of the case study. It takes some time for the tool to provide an answer (about 30 minutes) but finally it is able to generate a report with all the potential products of the SPL.

10.3.4.3 *Variability*

The FAMA tool was unable to provide an answer in 3 days, and the execution was aborted. As a consequence, we were unable to get the variability degree of the source model.

10.3.4.4 *Error detection*

This operation works well with the model of the case study. Answers were provided almost immediately providing the different model errors and their location. However, if it occurs in the variability

2 Windows 7 (x86). Pentium IV, 3.2GHz. 3GB RAM.

calculation, the tool is unable to retrieve the explanations of the errors.

10.3.4.5 *Core Features and Variant Features*

This kind of operation was not available for the model of the case study. The FAMA tool provided the following information message: *Current model does not accept this operation*.

10.4 SUMMARY AND CONCLUSIONS

In this chapter we have detailed the analysis process of an industrial feature model provided by Rolls-Royce using the validation capabilities provided by MULTIPLE.

First of all, we show step-by-step how to execute the tool to obtain the results. Furthermore, we have described and classified the errors found during the semantic and syntactical analysis of the model providing solutions.

The results provided the company a mechanism to enhance their large-scale feature model with the aim of integrating it in automated processes. Being conscious of the inconsistencies contained in the feature model, and having the analysis conclusions, they will be able to take the necessary steps to correct the feature model. With all that background, they will even be able to elaborate a set of good practices to follow when modifying the product line, reducing the number of errors incrementally.

On the other hand, the automated analysis allowed us to extract some important information of the model that we would have been unable to extract manually.

FAMA provides us with the mechanisms to keep track of errors in feature models. It is a powerful and useful framework that integrates the most important operations you need to analyse a feature model saving time in performing these tasks. Nevertheless, the use of FAMA with large-scale models reveals some tool limitations derived from

the presence of cross-tree relationships (*Requires* and *Excludes*) in such a big model.

The tool presents scalability problems in providing answers about some aspects of the input model, such as core and variant features, and error explanation. We have not been able to calculate the degree of variability of the model.

The analysis also revealed a model with many errors in its elaboration. The use of error checking mechanisms becomes essential in an industrial environment, where this model is being used as input to many other processes. Our work has revealed that it is possible to have a framework with enough power to perform a complete analysis (syntactic and semantic) of a large-scale feature model fitting different technologies together.

Part V

THE MULTIPLE FRAMEWORK IN 3RD PARTY PROJECTS AND TOOLS

SUMMARY

In this part we present some case studies in which the MULTIPLE framework has been used as a generic set of tools to support MDE processes. These case studies have been developed in collaboration with researchers of other universities and research groups. First, in chapter 11 we present the INTERGENOMICS case study, a work done in conjunction with the *Institut für Informationssysteme* at the *Technische Universität Braunschweig*. In this work the MULTIPLE framework is extended with additional metamodels and tools allowing to deal with biological models. Such models can be transformed to a formal specification, such as *petri nets*. This specification can be executed, and this way biological models can be validated. Second, chapter 12 presents a work done in the software measurement field. This work, has been done in collaboration with the *Alarcos* research group, *Universidad de Castilla–La Mancha*. In this work, the QVT–Relations transformations engine that the MULTIPLE framework provides is used to measure different software artifacts. Finally, in chapter 13 the MORPHEUS tool is presented. This tool aims to provide a tool to support the Architecture generaTed from RequIrements applying a Unified Methodology (ATRIUM) methodology. This work has been done in collaboration with E. Navarro, tenured assistant professor of the *Universidad de Castilla–La Mancha*, and the main researcher behind the ATRIUM methodology (Navarro 2007). In this work, the QVT-CLI transformations engine provided by MULTIPLE is used to generate software architectures.

11

INTERGENOMICS: BIOLOGICAL DATA MIGRATION USING THE MULTIPLE FRAMEWORK

«Nothing can be more incorrect than the assumption one sometimes meets with, that physics has one method, chemistry another, and biology a third»

— Thomas Henry Huxley
English biologist, 1825–1895

The traditional sequence of ‘‘experiment → analysis → publication’’ is changing to ‘‘experiment → data organization → analysis → publication’’ (Emmett et al. 2006). This is because, nowadays, data is not only obtained from experiments, but also from simulations. The great amount of new data that can be generated from these experiments is not always homogeneous and may be stored in different databases. Moreover, the quantity of data requires the development of new computer tools that allow us to represent, analyze, and make new simulations with them.

These problems are also found in the bioinformatics field, especially when analyzing and simulating cell-signaling mechanisms (*Signal Transduction Pathways*). A *signal transduction pathway* is a set of chemical reactions that occur inside the cell when it receives a stimulus. In studies of this type, it is very common to find both

independent databases and modeling tools. Thus, the data of the databases must be converted manually from the source databases to the simulation tools in order to be used. For this scenario, it is desirable to make interoperable tools available. It would certainly be beneficial to develop different models for a signal transduction pathway using different specification languages to be able to apply different simulation tools. However, this is only possible if the models do not have to be developed by hand but can be generated automatically from the source databases.

Collecting data from different sources in bioinformatics is a very common task. However data heterogeneity is a problematic issue. MDE techniques allow alleviating this problem by using model transformations.

Model Driven Software Development (MDSD) is an approach that attempts to solve problems of this kind. A model defines the functionality, structure or behaviour of systems (OMG 2003) depending on the metamodel used. As it has been largely discussed throughout this thesis, using models in a MDSD process allows the automation of the development and evolution of the software applications thanks to generative programming techniques (Czarnecki and Eisenecker 2000) such as model transformations and code generation.

This chapter shows how the MDSD philosophy can solve the problems that arise in the study of signal transduction pathways in the bioinformatics field. Problems like interoperability between applications can be addressed in a systematic way, where the data structure can be defined by using models and data is defined as a set of objects that are instances of the classes of these models. Dealing with data from the MDSD perspective helps to develop tools where the data processing mechanisms are independent of the final persistence format, obtaining more modular tools. This also helps to automate the data migration process by means of model transformation techniques. All these factors reduce the costs of the software development process, directly increasing the productivity of the users/biologists.

This chapter is organized as follows: section 11.1 explains the biological context, introducing the reason for studying the signal transduction pathways and describes the current approach, which is a very inefficient process. Section 11.2 describes the solution proposed to cover the shortcomings of the current approach. Section 11.3

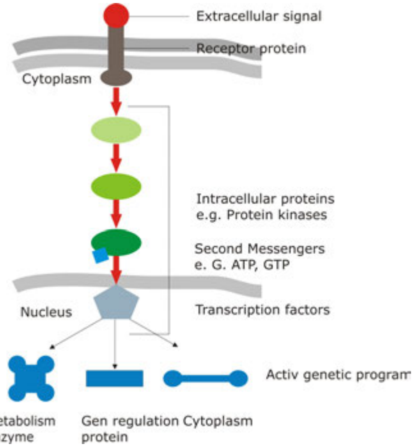


Figure 11.1: Signal Transduction, cf. (Alberts et al. 2005)

shows a running example of the solution proposed. Finally, section 11.4 presents the conclusions and future works.

11.1 CASE STUDY

In organisms, proteins have a wide variety of functions and they interact with each other in similar multifaceted ways. These interactions of proteins are described by means of signal transduction pathways or networks, which are typically represented as certain kinds of maps. A distinction is drawn between metabolic and regulatory pathways. Metabolic pathways describe the conversion of classes of substances into other classes of substances, whereas regulatory pathways describe how the function of something is regulated. In this case study, the conversion of classes of substances into other classes is not significant, but the transduction of signals is (cf. Fig. 11.1). That is why they are also called signal transduction pathways.

Signal transduction pathways are a representation of the chemical reactions that occur inside a cell when it is stimulated by an external event.

A signal transduction pathway describes how a cell responds to an extracellular signal, e. g. a signaling molecule excreted by a bacterium. The signaling molecule is received at a receptor protein and then transferred via biochemical reactions into the nucleus, where

it changes the behaviour that is currently active. Signal transduction pathways comprise different kinds of molecules: proteins and enzymes with different kinds of functions interact with the help of cofactors, second messengers, phosphatases and small effectors to transmit the signal. The mechanism of transmitting the signal is mediated through state changes of molecules like conformity changes and the building of molecule complexes on the basis of biochemical reactions. These molecule interactions cause the signal flow through the cell and the amplification of the signal in order to reach the nucleus. Figure 11.2 shows an example of a signal transduction pathway, where the gray area represents the inside of a cell and the light-colored area represents the outside. The nucleus is represented as a gray ellipse. In this map, molecules are represented with different shapes and colors, which encode the role that a certain molecule plays in the signal transduction pathway under consideration. Examples for such roles are extracellular signals, which are represented as stars; receptors, which are represented as rectangles across the cell membrane; and adapter proteins, which are represented as blue ellipses. Interactions of the molecules appear as lines and arrows, whereas their different shapes stand for different kinds of interactions, e. g. direct or indirect activation or inhibition. Molecules also interact by building molecule complexes, which are represented through narrow cumulations of molecules.

Signal transduction pathways are built in a collaborative way by experts worldwide. In the majority of the cases only small parts of these pathways are completely known.

These signal transduction pathways are composed by experts, who study the relevant literature that is produced by various groups worldwide doing research on very small parts of signal transduction pathways in different kind of organisms, e. g. research about short sequences of chemical reactions. This information is then composed bottom-up to a signal transduction pathway and introduced into databases to provide an integrated view on the entire, pathway.

Examples for such signal transduction pathway databases are TRANSPATH® (Krull et al. 2006), Kyoto Encyclopedia of Genes and Genomes (KEGG) (Kanehisa et al. 2006), Reactome (Joshi-Tope et al. 2005) and BioCyc (Karp et al. 2005). They usually provide a web interface for interactive searches and also make their data available

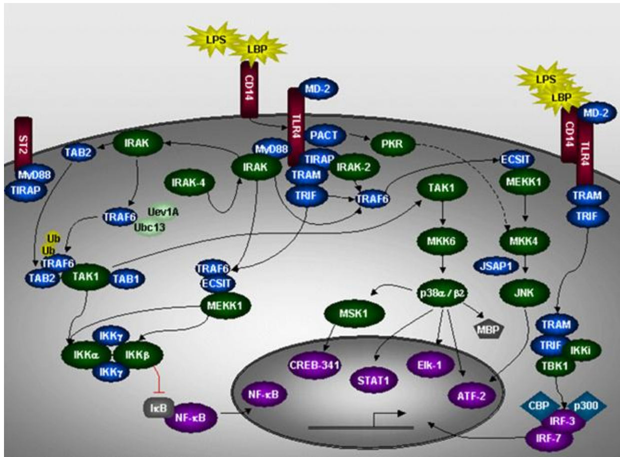


Figure 11.2: TLR4 signal transduction pathway in the TRANSPATH® database

as text files in flat file or XML format. Some of the databases already use a more or less standardized exchange format on XML basis, e. g. Systems Biology Markup Language (SBML) (Hucka et al. 2004).

11.1.1 *Toll-like receptors and the TLR4 signal transduction pathway*

In this case study we focus on the TLR4 signal transduction pathway, which is related with the immune system in mammalians. The TLR4 is one of the best known signal transduction pathways.

In order to give the reader a better understanding of what signal transduction pathways are about, we take the TLR4 signal transduction pathway as an example: Sepsis is the systemic immune response to severe bacterial infection (Motta and Brusic 2004). We are born with a functional, innate immune system that recognizes bacterial and viral products. In sepsis, when a bacterium attacks an endothelial cell, different kinds of mechanisms are activated. Receptors of the innate immune system are activated by microbial components such as the Lipopolysaccaride (LPS), an endotoxin which is a signaling molecule involved in the initiation of the sepsis syndrome. Receptors, which recognize such LPS molecules, are a family of transmembrane receptors known as Toll-Like Receptors (TLRs). To date, there are 12 TLRs identified in mice and 10 TLRs identified in humans.

TLR4 is one of these and is identified as a significant receptor in mice strain experiments. TLR4 is also annotated in TRANSPATH® and is our example for the explanation of signal transduction pathway (see Fig. 11.2). The TLR4 signal transduction pathway is subdivided into the MyD88-dependent and MyD88-independent pathway and consist of four chains. The whole pathway is specified in (Dauphinee and Karsan 2006). To give an overview over the general flow of information, it is sufficient to explain one part in detail.

One chain of the MyD88-dependent pathway in endothelial cells starts with the LPS binding to the TLR4 receptor complex consisting of CD14 and MD2. This molecule complex is leading to the recruitment of the adaptor molecules MyD88 and TIRAP. Following, IRAK and IRAK4 are recruited to the receptor complex via interaction of special parts of their spatial arrangements. IRAK recruits and activates TRAF6 which is one part of a molecule complex in addition consisting of ECSIT and MEKK1. This recruitment is leading to the activation of IKKα and IKKβ which are molecules of a complex with two IKKγ molecules. The activation of the IKK-complex leads to the degradation of Iκb. This inhibition of Iκb facilitates the translocation of NF-κB in the nucleus. NF-κB is a transcription factor which connects with its special promoter region. This results in the expression of proinflammatory molecules and furthermore normal physiological functions of the endothelial cells are perturbed. This bacterial sepsis and its associated expression of proinflammatory molecules causes death with the utmost probability.

In order to exemplify the transformation of signal transduction pathway data to Petri nets, we must first take a closer look at the biochemical reactions below that occur at the beginning of the signal transduction pathway: a signal molecule LPS arrives at the cell membrane and binds to the adaptor protein LBP (reaction 11.1) and is delivered to the receptor CD14 (reaction 11.2). This is the beginning of the signaling by TLR4 as mentioned above. The recruitment of the adaptor molecules MyD88 and TIRAP by the TLR4 receptor complex

can be inhibited by the sequestering of these critical adaptors during LPS signaling by ST2 (reactions 11.3 and 11.4, respectively).

$$\mathrm{LPS} + \mathrm{LBP} \leftrightharpoons \mathrm{LPS} : \mathrm{LBP} \tag{11.1}$$

$$\mathrm{LPS} : \mathrm{LBP} + \mathrm{CD14} \leftrightharpoons \mathrm{LPS} : \mathrm{LBP} : \mathrm{CD14} \tag{11.2}$$

$$\mathrm{ST2} + \mathrm{TIRAP} \leftrightharpoons \mathrm{ST2} : \mathrm{TIRAP} \tag{11.3}$$

$$\mathrm{ST2} + \mathrm{MyD88} \leftrightharpoons \mathrm{ST2} : \mathrm{MyD88} \tag{11.4}$$

11.1.2 *An approach to the study of the TLR4 signal transduction pathway*

Understanding the flow of information inside a cell is fundamental for an in-depth understanding of the functioning of a cell as a whole. Therefore, modeling and simulating this information flow is beneficial because it helps to understand the flow of signals in a complex network, to test hypotheses in silico before validating them with experiments, and to validate the data collected about a certain signal transduction pathway. The fact that a flow of information in a complex network must be described has led to the idea of applying languages for the description of concurrent reactive systems in this area, even if these were originally developed to assist the construction or engineering of systems and not the description of already existing systems (Fisher et al. 2004). A couple of specification languages, such as Petri Nets, Life Sequence Charts, etc., qualify for this task. All of them have different advantages and drawbacks. In the same way, the corresponding simulation tools have different strengths and weaknesses.

Signal transduction pathways have a concurrent nature. Thus, to simulate them by using formalisms for the study of concurrent software systems (such as petri nets*) is straightforward.*

We are currently working on one of the major signal transduction pathways databases, TRANSPATH® (Krull et al. 2006), and we are using Colored Petri Nets (Jensen 1997) among others (e. g. Life Sequence Charts (LSCs) (Damm and Harel 2001), UML-Statecharts, etc.) as the specification language. The corresponding simulation tool is *CPN Tools* (Jensen et al. 2007). TRANSPATH® is a database that is accessible by means of the usual methods, i. e., web interface, text files, XML (using its own XML format), etc. In January 2007,

TRANSPATH® contained entries about 60,000 molecules, 100,000 chemical reactions, 20,000 genes and 57 signal transduction pathways. The information was based on 30,000 publications. The web interface provides access to all these entries and also contains interactive maps, which give an overview of a certain signal transduction pathway (cf. Fig. 11.2). The XML version of the database is divided into six files containing data about molecules, genes, reactions, pathways, annotations and references, respectively. They are accompanied by Document Type Definition (DTD) describing the structure of the files.

Coloured Petri nets are a formal representation for distributed discrete systems that allow concurrent events to be represented. A Petri net consists of two types of nodes (*places* and *transitions*, respectively) and directed arcs. Arcs are always placed between transitions and places (or places and transitions). Places may contain any number of *tokens*. These tokens can be moved from one *place* to another when a transition is fired (the transitions are enabled if there are tokens in all their input places). Figure 11.3 shows an example of a Petri net. White circles represent the *places*, black rectangles represent the *transitions*, arrows represent the arcs, and large black dots represent the *tokens*. Coloured Petri nets are an enhancement of Petri nets and can contain different kinds of tokens identified by colors. Now it is possible to represent different dynamic behaviors modeled by different token colors in the same model. *CPN Tools* is a tool for constructing and analyzing coloured Petri nets.

In (Taubner et al. 2006), data is extracted from the TRANSPATH® database and introduced in the *CPN Tools* application manually. This

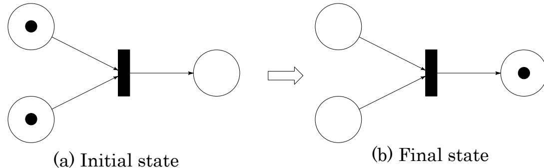


Figure 11.3: Petri net example

implies that the user/biologist who is going to perform the simulation must manually query the database to extract the list of reactions involved in the signal transduction *pathway* to be studied. With the extracted data the corresponding Petri net must be built in the simulation tool manually (creating each one of the places, transitions, arcs, tokens, etc. individually). In (Taubner et al. 2006), this means to manually defining approximately 75 places, 50 transitions, and 100 colours.

11.2 A MDSD APPROACH IN BIOLOGICAL DATA MIGRATION

In the initial work on the study of the TLR4 signal transduction *pathway*, data migration from the source database to the simulation tool (to represent this information as a coloured Petri net) was done manually.

The solution to the data migration problem is described as follows by means of model transformation techniques using the model-driven software development guides. This implies the following tasks: (i) development of the source domain data model (TRANSPATH®); (ii) development of the target domain data model (*CPN Tools*); (iii) definition of the transformation rules between the source domain and the target domain by means of the transformations language; (iv) implementation of the pre-processing mechanism to obtain the instances of the source model from the original data; and finally, (v) definition of the post-processing tasks that persist the transformed data in the final file format. The next subsections describe the designed solution. First, the transformation process and the different stages are described; second, the source and the target models are presented, and last, the transformation process is explained in more detail.

We propose to use declarative model transformations to perform the data migration process between different domains.

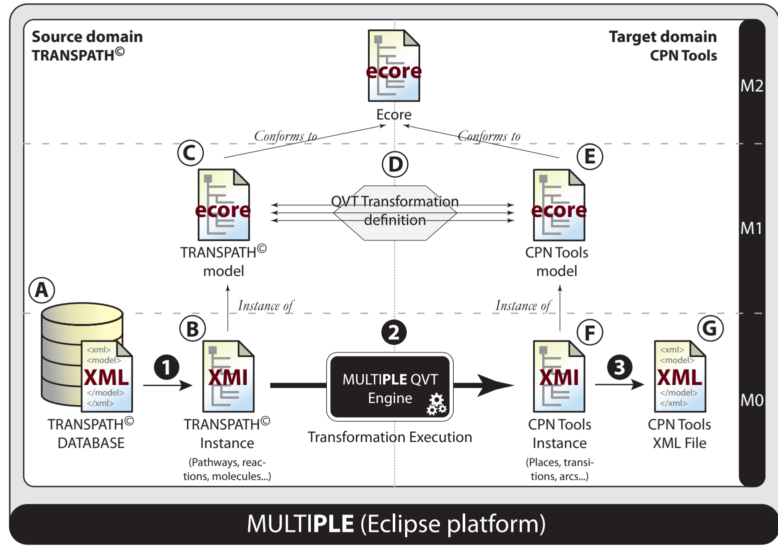


Figure 11.4: Architecture of the tool

11.2.1 *Architecture and overview of the tool*

The data migration process is performed in three steps: ❶ recovering and pre-processing of the input data, ❷ execution of the transformation by means of the transformations engine and ❸ post-processing and persistence of the result data. In a MDSD approach, using a transformation engine implies that the source and the target models of the transformation must be developed in the first place to be able to establish the mappings between the two domains.

The solution presented in this case study uses the MULTIPLE framework, which is integrated within the Eclipse platform and provides support for model transformations. This tool uses the Eclipse Modeling Framework (EMF), which provides *Ecore* as a modeling language. Moreover, it uses XMI as the persistence format and allows the creation of *Ecore* models from, among others, XSD.

Nevertheless, the *Ecore* models obtained from XSD schemas are complex and do not always clearly represent the real structure of the data. Therefore, the source and the target models have been defined

manually, taking into account only the information that is useful in the case study. Figure 11.4 shows the architecture of the tool. It represents the three steps that are needed to perform the data migration. First, the data is extracted from the TRANSPATH® database Ⓐ, and the corresponding XMI instance Ⓑ of the TRANSPATH® *Ecore* model Ⓒ is built. This first step ❶ is easily done in Java since the mappings between the elements of the source data and the elements of the EMF model can be established directly. The implementation of this pre-processing step has been adapted from the work done in (Ziegler 2007).

The second step ❷ is the most important and complex one of the transformation process. It is performed by means of the MULTIPLE tool and its transformation engine. It executes the transformation from the TRANSPATH® domain (reactions, molecules, etc.) to the *CPN Tools* domain (places, transitions, arcs, etc.). After the definition of the transformation rules Ⓓ between the source domain Ⓒ and the target domain Ⓔ, the transformation is executed over the data recovered from the database Ⓑ obtaining the needed information in the *CPN Tools* domain Ⓕ. Finally, the third step ❸ in the data migration process is again a trivial process in which the EMF data is stored in an XML file readable from the *CPN Tools* application Ⓖ. Other tasks can be performed in this stage; for example, the execution of some layout algorithms over the elements of the Petri net to represent the graphical elements properly in the drawing space of the *CPN Tools* GUI.

11.2.2 *Development of the source and the target models*

First, a model that contains the most interesting elements to simulate a signal transduction pathway in the *CPN Tools* application has been defined (removing unnecessary concepts from the complex TRANSPATH® database).

Using a visual metaphor similar to the UML class diagram, Figure 11.5 shows the *Ecore* model that has been developed. The *Network* class is the main element in this model. A *Network* contains

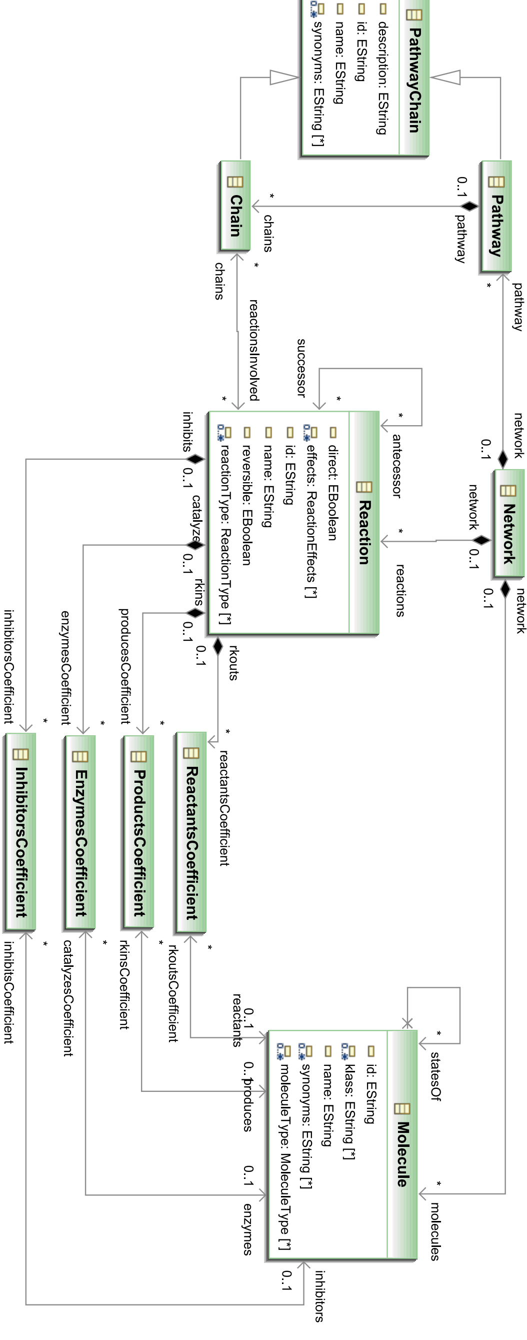


Figure 11.5: Model of the TRANSPATH® database

a set of *Pathways*, *Reactions* and *Molecules*. Moreover, a *Pathway* is composed of several *Chains* of *Reactions*, and one *Reaction* can be involved in several *Chains*. Finally, the reactions are related to the molecules. One molecule can be a reactant or a product in a reaction. It can also take part in a reaction as an inhibitor or a catalyst (if the molecule is an enzyme). The classes *ReactantsCoefficient*, *ProductsCoefficient*, *EnzymesCoefficient* and *InhibitorsCoefficient* inherit from the *Coefficient* class (omitted for reasons of clarity). This class contains an integer attribute (*coefficient*[1]).

Figure 11.6 shows the model that has been created for the *CPN Tools* application. In this case, the design of the model is closer to the application specific concepts than to the conceptual Petri net concepts. This design allows us to deal with all the interesting concepts of the *CPN Tools* platform (e. g., position and color of the graphical elements). Furthermore, this kind of design makes the persistence process from EMF to the final XML file easier.

We have defined the metamodels for the source and the target domains. The correspondences between these spaces can be defined by using patterns.

Cpnet is the main class of the mode (see Figure 11.6). It is divided, by using a dashed line, into two groups: the classes that are under the *Globbox* element and the classes under the *Page* element. The first group (the *Globbox* group) allows the declarations of CPNs such as colorsets (enumerated, complex), variables, blocks, etc. The second group of classes (those contained in the *Page* element) represents all the visual elements of the coloured Petri net. All the graphical elements inherit from the *DiagramElement* class, and can be contained in different groups (*Group* class). Thus, a *Page* can hold *Places*, *Trans* (transition), *Arcs*, *Annot* (annotations), etc. When a *Place* is defined in the Petri net, it has an associated color set. This color set must be defined previously in the declarations part. The relationship between the *Place* and its color set is represented by means of the *type* role from the class *Place* to the class *ColorSet*. The classes *InitMark* and *Mark* are intended to represent the actual state of a given

1 The coefficient value represents the number of molecules that appear in the equation of one reaction. For example, in the reaction $2H_2 + O_2 = 2H_2O$, the coefficients are the numbers that appear on the left of the molecules, i. e., $\underline{2}H_2 + \underline{1}O_2 = \underline{2}H_2O$

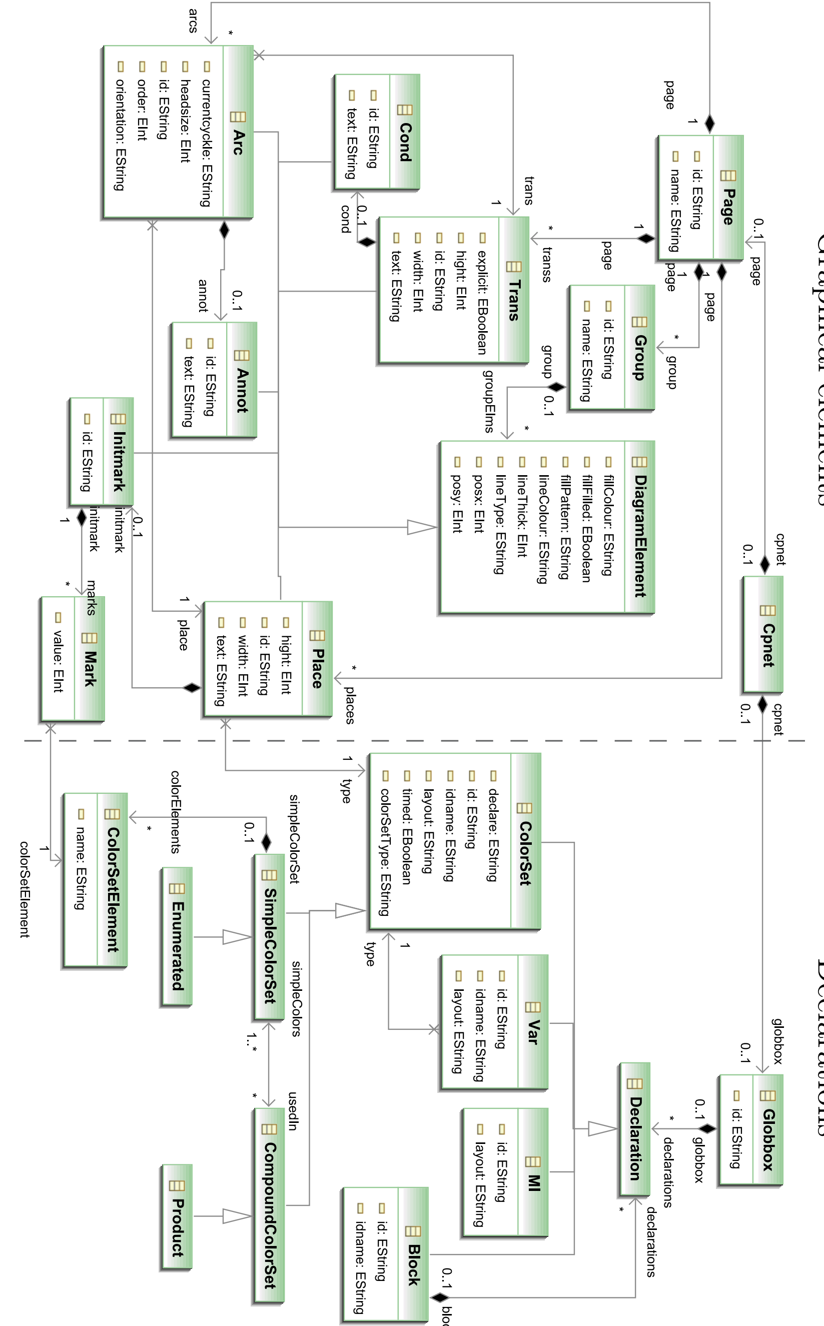


Figure 11.6: Model of the *CPN Tools* application

Place, indicating which tokens are in the *Place*. The kind of tokens is defined by the role *colorSetElement* between the classes *Mark* and *ColorSetElement*.

11.2.3 *Transformation process*

Finally, the transformation rules that can convert data from the source domain to the target domain have been defined. These rules express the mappings established by biologists between the data extracted from the TRANSPATH® database and the concepts available in the *CPN Tools* application. Table 11.1 shows the simplified mappings between both the source and the target domain. The rules that define the direct relationships between the two domains have been expressed in QVT–Relations.

The transformation is executed as a top-down process. The navigation is performed through the containment relationships (defined in *Ecore* by means of containment references), i. e., it begins from the root element of the source model (*Network*) and goes down (*Network*

TRANSPATH	CPN TOOLS
Network	Cpnet
Pathway	Globbox, Page
Molecule (complex)	Product
Molecule (simple)	Enumerated
Reaction	Trans
Molecule (reactant)	Place, Arc (from Place to Trans)
Molecule (product)	Place, Arc (from Trans to Place)

Table 11.1: Mappings between the source and the target domain

→ *Pathways* → *Chains* → *Reactions* → *Coefficients* → *Molecules*) creating the corresponding elements in the target domain as Table 11.1 defines. In the declarations group, the transformation will create an *Enumerated ColorSet* from each simple molecule. In the case of the complex molecules, the *ColorSet* created will be a *Product*. This *Product* will be be a compound of the *Enumerated ColorSets* corresponding to the simple molecules which are part of the complex molecule.

In the graphical elements group, the transformation process begins from a *Reaction* element. An object of the class *Trans* is created for each reaction. We obtain the reactant molecules through the *reactantsCoefficient* association in the class *Reaction*. A *place* is created for each one of these molecules. Finally, each new *place* can be linked with its corresponding *Trans* element by means of an *Arc*. These *arcs* will be of type *PtoT* (*Place to Trans*, according to the *CPN Tools* terminology). The procedure is similar for the products of the reactions; however in this case, the transformation navigates through the *productsCoefficient* association.

Figure 11.7 represents the result of the transformation process (in the *CPN Tools* metaphor) for the reactions presented in the case study. The figure shows four numbered triangles, each of which corresponds to one of the reactions of the example. Thus, for reaction number 11.1 ($\mathsf{LPS} + \mathsf{LBP} \leftrightharpoons \mathsf{LPS} : \mathsf{LBP}$) the transformation generates the elements inside the left triangle (1). The other three triangles (2, 3, and 4) indicate the corresponding reactions (11.2, 11.3 and 11.4).

Appendix G lists the textual representation of the *Transpath2CPN* transformation.

11.3 RUNNING EXAMPLE

This section shows a running example using the example data presented before. This way, the demonstration will use two initial files:

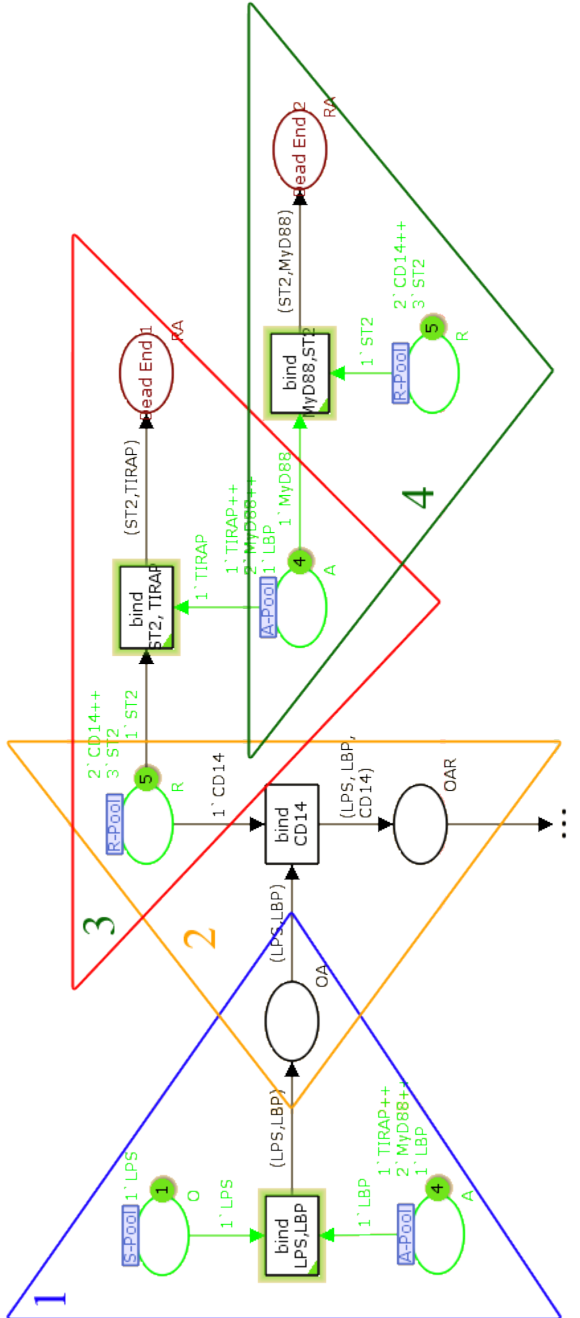


Figure 11.7: Partial representation of the TLR4 signal transduction pathway in *CPN Tools*

example.xml — An XML file containing the data extracted from the TRANSPATH® database. This file contains information about a single pathway (the TLR4 pathway).

transpath2cpn.qvt — This file contains the Transpath2CPN transformation.

Fig. 11.8 shows the example workspace with these two files. The `example.xml` file is shown in its default editor, i. e., the TRANSPATH® model editor.This editor is able to represent the information extracted from TRANSPATH® as an instance of the *transpath metamodel*. Fig. 11.9 shows the actual contents of the `example.xml` file. As can be observed, `example.xml` is an XML file which has been directly extracted from the TRANSPATH® database.

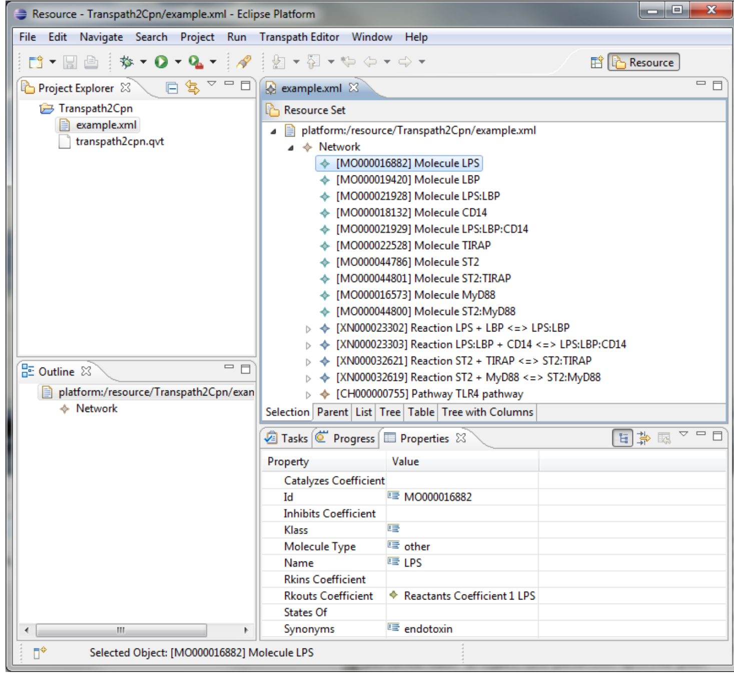


Figure 11.8: Workspace with the example files

example.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<network extent="pathway" source="bbdb101" version="7.1" xmlns:xlink="http://www.w3.org/19
    <Pathway id="CH000000755">
        <!-- Copyright (c) Biobase GmbH -->
        <creator>mkl</creator>
        <updator>mkl</updator>
        <name>TLR4 pathway</name>
        <synonyms>LPS pathway; lipopolysaccharide/endotoxin pathway; .
        </synonyms>
        <type>pathway</type>
        <chains>
            <PathwayReference id="CH000000569"/>
            <PathwayReference id="CH000000571"/>
            <PathwayReference id="CH000000572"/>
            <PathwayReference id="CH000000831"/>
        </chains>
        <pathways>
```

Figure 11.9: Actual contents of the `example.xml` file

To execute the model transformation the user can use the *Run as* → *1 QVT Transformation* menu, as it was shown in section 8.4.4.2. To execute a model transformation, the different domains of the transformation must be set, as shown in Fig. 11.10.

11.3.1 *Result files*

Once the transformation has been executed, two new files appear in the workspace. The former corresponds with the result model

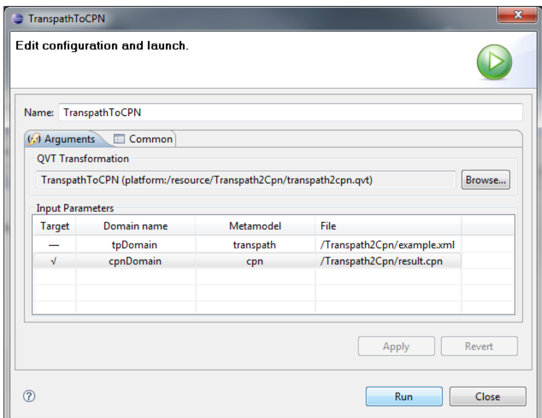


Figure 11.10: Transpath2CPN transformation ready to be executed

(`result.cpn` in this example), and the latter corresponds with the traces model (`result.traces`). Fig. 11.11 shows the Explorer view with the result files highlighted.

The result file is an XMI document, which is instance of the *CPN Tools metamodel*. Fig. 11.12 shows the `result.cpn` file in the *Cpn model editor*. The traces model can also be opened in the default traces editor, which was shown in section 8.4.3.

To be able to open the result model in *CPN Tools*, it is necessary to convert the result file to a new XML document first. This new XML file adheres to the XSD defined by *CPN Tools*. This projectio step can be done using the contextual menu shown in Fig. 11.13. Moreover, at this point it is possible to execute a layout algorithm (if this step has not been previously executed within the QVT model transformation).

Fig. 11.14 shows the contents of the final XML in the Eclipse default textual editor. As it can be observed, this is a valid document that can be opened by *CPN Tools* directly.

11.3.2 *Result file in* CPN Tools

Finally, Fig. 11.15 shows what the petri net looks like in the *CPN Tools* interface. The position of the transitions and places can vary depending on the result of the layout algorithm because it is non-deterministic. It is not necessary to modify the petri net in any way in order to perform simulations using the *Simulation* toolbox.

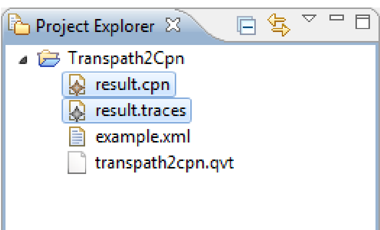


Figure 11.11: Result files

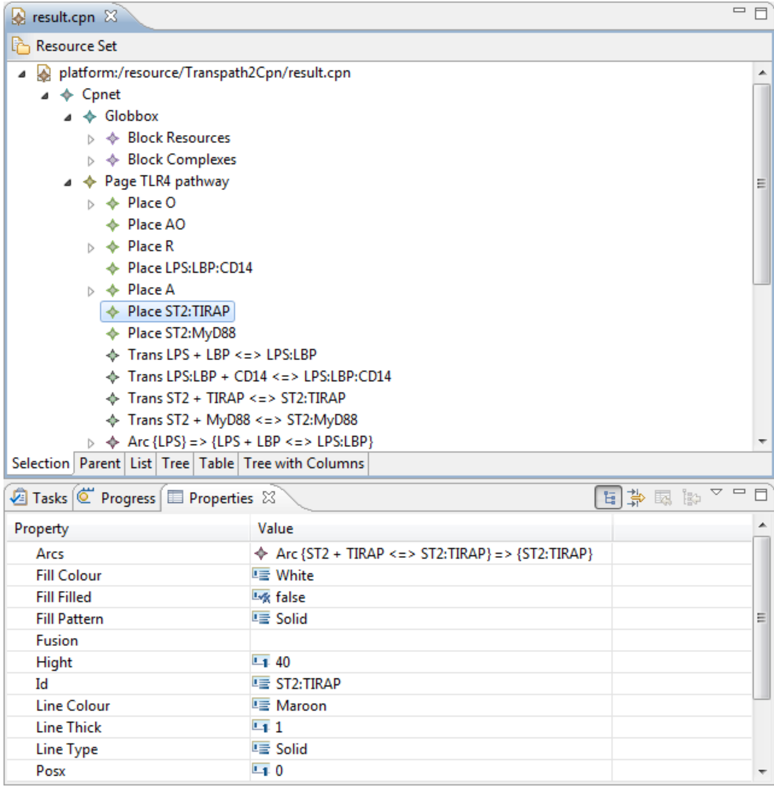


Figure 11.12: Editor for *cpn* models

11.4 CONCLUSIONS

This chapter has presented a case study where the interoperability problem between bioinformatic applications is addressed using a model-driven approach. The situation where several data sources and simulation tools co-exist and must share heterogeneous data is very common in the bioinformatics field. In this situation, the easy representation of biological data using models allows us to deal with these problems more efficiently and more elegantly than the traditional (manual) approaches. It is more efficient because

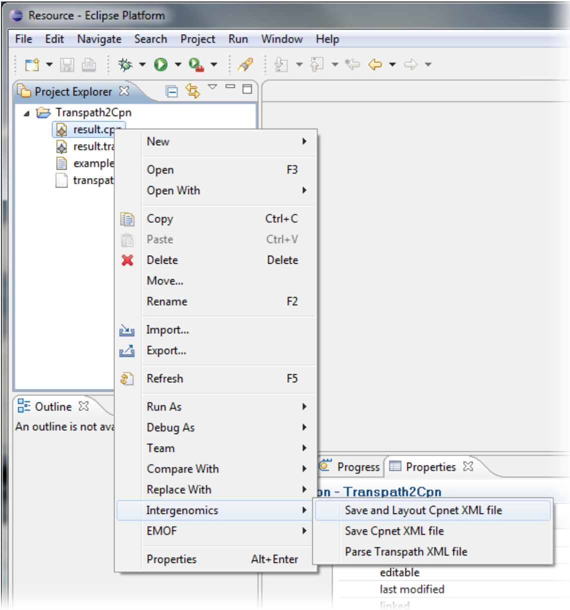


Figure 11.13: Export to *CPN Tools*

the software development process is shorter. It is more elegant because the operations are done at a higher level of abstraction and the language used is more expressive due to its declarative nature. In (Garwood et al. 2006; Bhattacharya et al. 2005; Komatsoulis et al. 2007; Li et al. 2006; Song et al. 2007) also model-driven approaches are applied in the life sciences but not in the field of signal transduction pathways. This work presents the following advantages over the traditional approaches: (i) It allows some tasks that were previously done by hand to be automated. (ii) This approach produces more modular tools, making the transformation mechanism independent from the data persistence format, improving the extensibility and maintainability of these tools. (iii) Biologists do not need to know technical details about the migration process, which increases their productivity. (iv) It also takes advantage of model transformation technologies. Using models to represent the data to be transformed

```
result.xml
<?xml version="1.0" encoding="iso-8859-1"?>
<workspaceElements>
  <generator format="5" tool="CPN Tools" version="2.0.0"/>
  <cpnet>
    <globbox>
      <block id="ID18027118">
        <id>Resources</id>
        <color id="ID18812666">
          <id>O</id>
          <enum>
            <id>LPS</id>
          </enum>
          <layout>colset O = with LPS;</layout>
        </color>
        <color id="ID1080367">
          <id>A</id>
          <enum>
            <id>TIRAP</id>
            <id>MyD88</id>
            <id>LBP</id>
          </enum>
```

Figure 11.14: Content of the final XML file

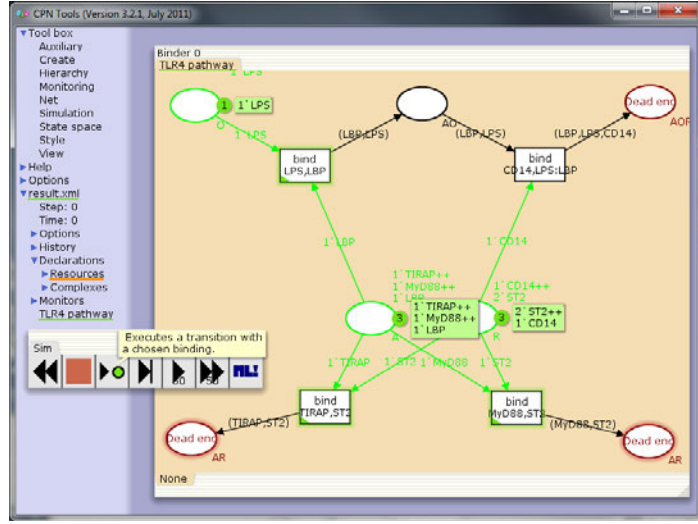


Figure 11.15: Final result shown in *CPN Tools*

permits the data structure to be more clearly represented making its manipulation more intuitive since it deals with high-level concepts. (v) Traceability capabilities are provided implicitly. These capabilities help to locate invalid information in the data sources. Finally, (vi) using languages such as QVT–Relations offers the advantage of expressing the mappings between the source and the target domains in a declarative way. This way of representing the relationships between the two domains is more expressive than the traditional and imperative approaches. With this case-study we have presented the first steps in using model-driven techniques in the live science, which in the future can lead us to automatically generate more efficient and attractive visual metaphors and tools.

12

INGENIO: SOFTWARE MEASUREMENT BY USING QVT TRANSFORMATIONS IN AN MDA CONTEXT

«One accurate measurement is worth a thousand expert opinions»

— Grace Murray Hopper
American computer scientist and United States Navy officer, 1906–1992

The current necessity of the software industry to improve its competitiveness forces continuous process improvement. This must be obtained through successful process management (Florac et al. 2000). Measurement is an important factor in the process life cycle due to the fact that it controls issues and lacks during software maintenance and development. In fact, measurement has become a fundamental aspect of Software Engineering (Fenton and Pfleeger 1998).

Software Processes constitute the work base in a software organization. Companies therefore wish to carry out an effective and consistent software measurement to facilitate and promote continuous process improvement. To do this, a discipline for data analysis and measurement (Dennis and Goldenson 2004), and measure definition, compilation and analysis in the process, projects and software products, is needed.

The great diversity in the kinds of entities which are candidates for measurement in the context of the software processes points to the importance of providing the means through which to define measurement models in companies in an integrated and consistent way. This involves providing companies with a suitable and consistent reference for the definition of their software measurement models along with the necessary technological support to integrate the measurement of the different kinds of entities.

With the objective of satisfying the exposed necessities, it is highly interesting to consider the MDE paradigm (Bézivin et al. 2005) in which Software Measurement Models (SMM) are the principal elements of the measurement process, so that designs are expressed and managed a much higher level of abstraction.

MDE provides a suitable basis to represent software artifacts with a high level of abstraction and precision. This way, software measurement experts can benefit of the current standards and tools.

Software measurement can benefit from the MDE paradigm, providing integration and support to carry out an automatic software measurement of any software type. This implies that: (i) the definition of measurement models conform to a Software Measurement metamodel; (ii) the definition of generic measurement methods are applicable to any model-based software artifact; and (iii) support for computing measures, for storing results and for enhancing decision making.

These aspects constitute the main interest of the work presented in this chapter, in which the application of MDA principles, standards and tools are used in software measurement. The goal of this proposal is to develop a generic framework to define measurement models which conform to a common measurement metamodel, and to measure any software entity with regard to a domain metamodel. In order to develop this proposal, the MULTIPLE framework which has been presented in chapter 8 has been used.

Some publications (García et al. 2006; García et al. 2005; Garcia et al. 2007) are used as a starting point for this work. These works present Framework for the Modeling and Evaluation of Software Processes (FMESP), which consists of a framework based on MOF and MDA. This includes a software measurement ontology and metamodel, and the *GenMETRIC* tool which is used to define software

measurement models, and to calculate defined measures for these models. The ontology permits the identification of all the concepts, proportions exact definitions for all the terms and clarifies the relationship between them. This chapter presents an adaptation of FMESP to MDA, which is described in detail in following sections.

The remainder of the chapter is organized as follows. Section 12.1 provides an overview of related works and Section 12.2 describes the SMF, including conceptual architecture, technological aspects, and method. In Section 12.3 the use of the framework is illustrated with an example. Finally, conclusions are outlined in Section 12.4.

12.1 RELATED WORKS

We have found numerous publications which deal with tools that have important success factors in software measurement efforts (Komi-Sirviö et al. 2001), which supply work environments and general approximations (Kempkens et al. 2000), or which give architectures more specific solutions (Jokikyyny and Lassenius 1999). Dennis and Goldenson (2004) give a list of tools which support the creation, control and analysis of software measurements. (Auer et al. 2003) furthermore examines various software measurement tools, such as *MetricFlame*, *MetricCenter*, *Estimate Professional*, *CostXPert* and *ProjectConsole*, in heterogenic environments.

It is also possible to find certain proposals through which to tackle software measurement which are more integrated and less specific than in the aforementioned cases. Palza et al. (2003) propose the Multidimensional Measurement Repository (MMR) tool which is based on the Capability Maturity Model Integration (CMMI) model for the evolution of software processes, and it is possible to consult similar tools in (Harrison 2004; Lavazza and Agostini 2005; Scotto et al. 2004). These proposals are, however, restricted to concrete domains or to evaluation models of specific quality.

Vépa et al. (2006) present a metamodel which allows the storage of measurement data, and a set of transformations through which to carry out the measurement of models based on a metamodel.

This work focuses upon the technological aspects needed to implement the software measurement with ATL technology, by offering the user a variety of graphic representations of the measurement results obtained.

This final proposal and the one presented here are complementary as they both focus upon two key support elements of generic measurement: the conceptual base, which is the main contribution of FMESP, and technological implementation. Some differences from technological point of view exist.

The measurements which are applied by Vépa et al. (2006) are previously defined in the ATL transformation archives. The measurable entities are typical of the metamodels presented in this work (Kernel Meta Meta Model (KM3) and UML2). For example, the measurable entities for a model which is expressed in KM3 might be package, class, attribute, reference etc.

The measurements in the proposal presented here are defined by the user, i. e. the model transformation needed to carry out the measurement it is not a model previously defined, but this model is defined according to the users needs. The measurement definition is possible thanks to the software measurement model, which contains all that is relative to the measurement to be carried out in each case. Moreover, the measurable entities are those which are defined in their corresponding domain and measurement metamodel (expressed in *Ecore*). A further difference is that SMF uses QVT.

12.2 SOFTWARE MEASUREMENT FRAMEWORK

In order to carry out this proposal it was considered of interest to adapt FMESP to the MDE paradigm. The objective of this was to exploit the benefits that the paradigm could contribute to software measurement by, on one hand adopting the software measurement metamodel defined in FMESP, and on the other by evolving *GenMETRIC* to an environment which would allow the definition of software measurement models and the computation of the models defined. All this would take place within the context of models and

model transformations of the MDA architecture. The SMF is the evolution of the FMESP, but is adapted to the MDE paradigm and uses MDA technology.

The following subsections explain the conceptual, technological and methodological elements which are part of SMF.

12.2.1 *Conceptual architecture*

Due to the necessity of having a generic and homogeneous environment for software measurement (García et al. 2006; García et al. 2005; Garcia et al. 2007), a conceptual architecture and a tool with which to integrate the software measurement are proposed. In the following section, the main characteristics of this proposal are described. A more detailed description can be found in (Garcia et al. 2007).

The proposed software measurement described in this chapter is part of the FMESP framework (García et al. 2006). The FMESP framework permits representing and managing software processes from the perspectives of modeling and measurement. We focus on the measurement support of the framework whose elements are detailed according to the three layers of abstraction of metadata that they belong to, according to the MOF standard. In Fig. 12.1, the conceptual architecture for integrated measurement is represented.

FMESP is a framework which permits representing and managing software processes from the perspectives of modeling and measurement.

As can be observed in Fig. 12.1, the architecture has been organized into the following conceptual levels of metadata:

Meta-metamodel Level (M3) — At this level, an abstract language for the definition of metamodels, is found. This is the MOF language.

Metamodel Level (M2) — In the M2 level, two generic metamodels which conform with this framework are required. These are: the *Measurement Metamodel*, to define specific measurement models; and *Domain Metamodels*, to represent the kinds of entities which are candidates for measurement in the context

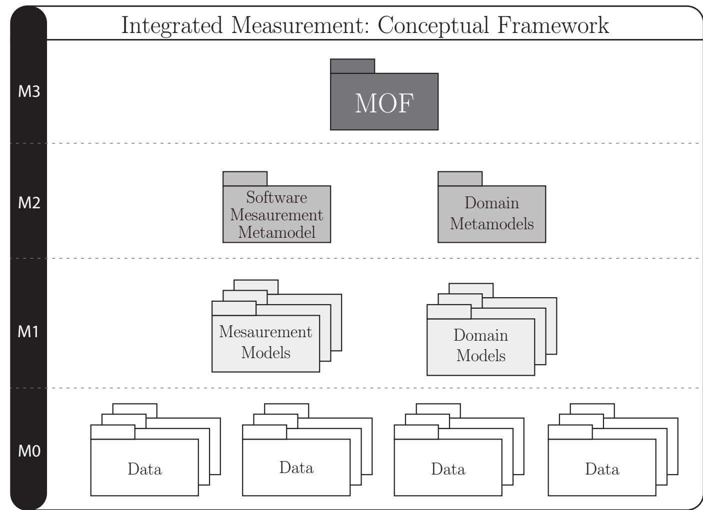


Figure 12.1: Conceptual framework with which to manage software measurement

of the evaluation of the software processes, such as, UML and Process metamodels.

*Model Level (*M1*)* — Specific models are included at this level. These models may be of two types: *Measurement Models*, which are examples of the measurement metamodel in the M2 level and which are defined in such a way as to satisfy some of the company's information needs; and *Domain Models*, which are defined according to their corresponding domain metamodels.

In order to establish and clarify the concepts and relationships that are involved in the software measurement domain before designing the metamodel, an ontology for software measurement was developed (García et al. 2005). The measurement metamodel was derived by using the concepts and relationships stated in the ontology as a base. The Software Measurement metamodel (which is integrated in SMF) is organized around four main packages—for greater detail see (García et al. 2005):

Software Measurement Characterization and Objectives — which includes the constructors required to establish the scope and objectives of the software measurement process.

Software Measures — which aim at establishing and clarifying the key elements in the definition of a software measure.

Measurement Approaches — This package introduces the *measurement approach* element which is used to generalize the different approaches used by the three kinds of measures to obtain their respective measurement results. A base measure applies a measurement method. A derived measure uses a measurement function. Finally, an indicator uses an analysis model to obtain a measurement result that satisfies an information need.

Measurement Action — This establishes the constructs related to the act of measuring software. A measurement (which is an action) is a set of measurement results, for a given attribute of an entity, using a measurement approach. Measurement results are obtained as the result of performing measurements (actions).

The Software Measurement metamodel is organized in four packages: Software Measurement Characterization and Objectives, Software Measures, Measurement Approaches *and* Measurement Action.

12.2.2 *Technological aspects*

In this section the technological aspects of SMF are explained.

12.2.2.1 *Adaptation to MDA*

In Fig. 12.2 the necessary elements for the FMESP adaptation to MDA are presented according to MOF levels.

As can be observed in Fig. 12.2, two new elements, namely the QVT–Relations Model and metamodel, have been added to adapt the conceptual architecture illustrated in Fig. 12.1 to MDA. The QVT–Relations Model (which is described in greater detail in Section 12.2.2.2) is obtained automatically through a transformation from

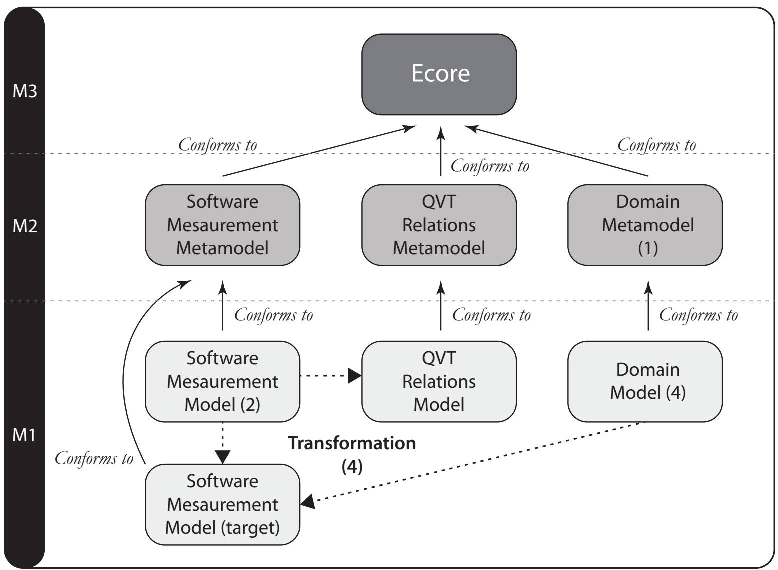


Figure 12.2: Elements of the FMESP adaptation in a MDA context

a Measurement model. It contains all the information necessary to carry out the transformation of the SMF proposal. *Ecore* language has been selected because it is a common and widely used modeling language based on EMOF as it has been demonstrated throughout this thesis.

12.2.2.2 *QVT–Relations transformation*

The QVT–Relations model is the transformation needed to perform the measurement. In this transformation two source models are involved: a Software Measurement model and a domain model; the target model is the Software Measurement Model with the measurement results (see Fig. 12.2). Due to the fact that the proposal is about generic measurement, it is very important that the QVT model is obtained in a generic way. The MDE paradigm and MDA technology are applied for this reason.

This transformation is obtained automatically from the previous QVT transformation shown in Fig. 12.3. The QVT–Relations model,

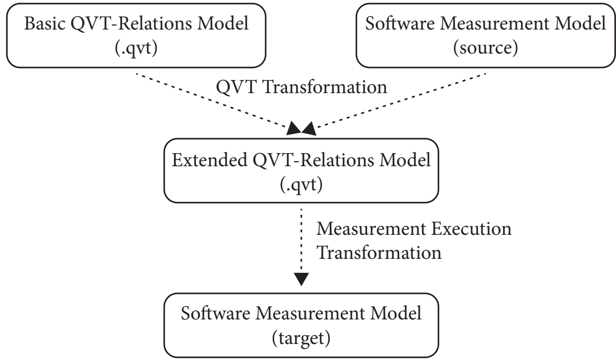


Figure 12.3: QVT–Relations transformation model

called the extended or final QVT–Relations model, is obtained from a QVT transformation, where there are two source models: the basic or initial QVT–Relations model (which conforms to the QVT–Relations metamodel) and the SMM previously defined.

The extended QVT–Relations model extends the basic QVT–Relations model with the following aspects:

In the MDA adaptation of FMESP, the QVT–Relations language is represented as a MOF artifact. This way, QVT transformations can be refined using QVT itself. In FMESP a Software Measurement Model *is combined with a* Basic QVT Transformation Model *to obtain a* Extended QVT Transformation Model.

- *Transformation Model.* To obtain the extended QVT–Relations model, the source model specification is needed. In this case, there are two source models: the SMM and the domain model. Due to the fact that the SMM is always the same, this model is already defined in the basic QVT–Relations model. Therefore, only the domain model needs to be defined. This information is taken from the Software Measurement model which contains all the measurement information.

- *Relation Domain.* In order to perform the transformation, it is necessary to define the *checkonly domain* object templates. In this case there are two, one for each source model: the domain model and the SMM.

- *Function.* This element contains the necessary OCL queries to carry out the measurement. These OCL queries are the

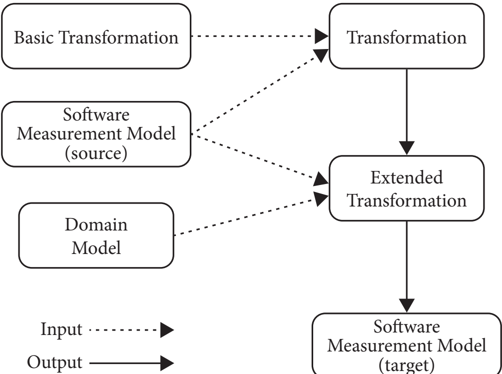


Figure 12.4: Software Measurement Process

implementations of the *Measurement Action* package defined in the Software Measurement Metamodel.

These elements are empty in the basic QVT–Relations model, and they are extended to obtain the extended QVT–Relations model, the transformation model necessary to carry out the measurement. In the Fig. 12.4 all the software measurement process is shown.

12.2.3 *Method*

The necessary steps to carry out the software measurement by using the SMF are explained below (see Fig. 12.2):

1. *Incorporation of domain metamodel.* The measurement is made in a specific domain. This domain must be defined according to its metamodel (it is situated in the M2 level and it conforms to the *Ecore* meta-metamodel).
2. *Creation of measurement model.* The measurement model is created according to the Software Measurement metamodel which is integrated in SMF. This first model is the source model, so the results are therefore still not defined, i. e., the

Measurement Action package from the Software Measurement metamodel is still not instantiated.

3. *Creation of domain model*, which is defined according to its corresponding domain metamodel (created in the first step). The domain models are the entities whose attributes are measured by calculating the measurements defined in the corresponding measurement models. Examples of domain models are: the UML models (use cases, class diagrams, etc.), or the E/R models.

4. *Measurement execution*. the measurement execution is carried out through QVT transformation, in which, the measurement model is obtained by starting from the two source models (the measurement model and the domain model) where the results are defined, i. e., the *Measurement Action* package is instantiated. The target measurement model is the extension of the source measurement model. The measurement results are calculated by running OCL queries on the domain model.

An example of the method application is shown in the following section.

12.3 EXAMPLE

To illustrate the benefits of the proposal, consider the example of relational database measurement. For greater simplicity, only the following elements are shown in Fig. 12.5: *Measurement Method*, *Entity* (to which the measurement method is applied) and *Measurement result* (the result is obtained by executing the measurement method on the entity).

Furthermore, it is necessary for the domain metamodel, in this case Relational Databases domain, to have been previously chosen. Both metamodels are independent (Fig. 12.5), although they are logically related. In Fig. 12.5 the measurement and domain metamodels have been represented in different colours.

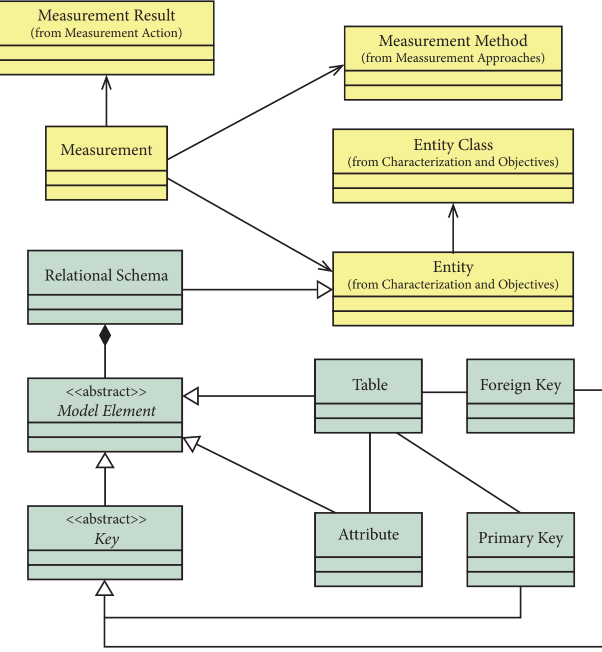


Figure 12.5: Relationship between Relational Database (domain) Metamodel and SMM

In this example, the chosen measurement method has been *COUNT elements of type* TABLE, which is an instantiation of the abstract method *COUNT elements of type* X.

In order to carry out the measurement, the following four steps must take place:

1. Incorporation of Relational Databases metamodel (represented in a dark colour in Fig. 12.6).

2. Creation of measurement model conforms to Software Measurement metamodel. For the measurement method *COUNT elements of type* TABLE, the values of *Entity* and *Measurement Method* are *Table* and *Count*, respectively. The *Measurement Result* is not still defined.

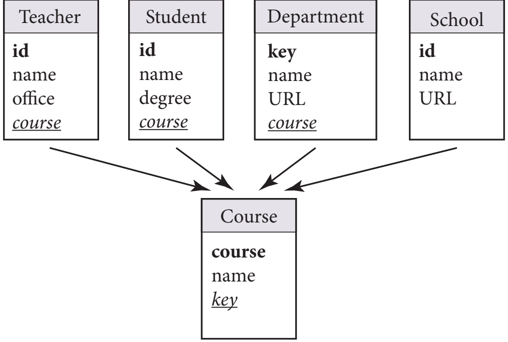


Figure 12.6: Relational Database model (relational schema)

3. Creation of model conforms to the Relation Database metamodel. In this case, the model (relational schema) is a university domain composed of five tables with their corresponding primary keys (bold), foreign keys (underlined and italic), and attributes (see Fig. 12.6).

   The extended QVT–Relations model is needed to carry out the fourth step. This transformation is obtained automatically (see section 12.2.2.2). The extended elements are detailed below:

   Transformation Model — The target model is the relational databases domain model.

   Relation Domain — The checkonly domain of the relational schema domain is indicated (see Listing 12.1).

   Function — this contains the OCL queries with which to perform the measurement, in this case, the queries necessary to implement the *COUNT element of type* X measurement method where X is `Table` (see Listing 12.2).

4. The source models used to carry out the measurement are: the measurement model (second step), the domain model (third step) and the extended QVT–Relations model. The target model obtained is the measurement model with defined

Listing 12.1: Relation domain elements from extended QVT–Relations model

```
1   checkonly domain relationDomain srcRelationalSchema :
        RelationalSchema {
2     name = myRelationalSchema
3   };
4   checkonly domain measurementDomainSrc srcMeasurementModel :
         MeasurementModel {
5     modelName = myModelName,
6     measurements = dstMeasurement1 : Measurement {
7       name = myMeasurementName,
8       method = dstMethod : MeasurementMethod {
9         nameMethod = myMethod
10      },
11      entity = dstEntity : Entity {
12        nameEntity = myEntity
13      },
14      result = dstResult : MeasurementResult {}
15    } // Result not defined yet
16  };
```

Listing 12.2: Function elements from extended QVT–Relations model

```
1 // [...]
2   enforce domain measurementDomainDst dstMeasurementModel :
        MeasurementModel {
3     modelName = myModelName,
4     measurements = dstMeasurement1 : Measurement {
5       name = myMeasurementName,
6       method = dstMethod : MeasurementMethod {
7         nameMethod = myMethod
8       },
9       entity = dstEntity : Entity {
10        nameEntity = myEntity
11      },
12      result = measurementAction(srcRelationalSchema,
              myMethod, myEntity)
13    }
14  };
15 } // End of relation
16 function measurementAction(relationalSchema :
      RelationalSchema, method : String, entity : String) :
      Integer {
17  relationalSchema.modelElements->select(
18        m : ModelElement | m.oclIsTypeOf(Table))-> size()
19 }
```

Listing 12.3: Measurement result

```
1  <?xml version="1.0" encoding="ASCII"?>
2  <measurement:MeasurementModel xmi:version="2.0"
3      xmlns:xmi="http://www.omg.org/XMI"
4      xmlns:measurement="http://bmora/metamodels/measurement"
5      modelName="ER_MEASUREMENT">
6    <measurements name="RELATIONAL SCHEMA MEASUREMENT">
7      <method nameMethod="COUNT"/>
8      <entity nameEntity="TABLE"/>
9      <result result="5">
10   </measurements>
11 </measurement:MeasurementModel>
```

Measurement Result (see Listing 12.3). In this example the value of *Measurement Result* is 5 (number of tables).

In the same way as is illustrated with Relational Databases, the method can be applied to any other domains, such as for example, UML models, Project Management or Business Processes, etc.

12.4 CONCLUSIONS

In this chapter a generic framework for the definition of measurement models based on a common metamodel has been presented. The framework allows the integrated management and measurement of a great diversity of entities.

Following the MDA approach and starting from a (universal) measurement metamodel, it is possible to carry out the measurement of any domain by means of QVT transformation, and this process is completely transparent to the user.

With SMF, it is possible to measure any software entity. The user task consists in selecting the domain metamodel (the domain to be measured) and defining the source models. The software metamodel is integrated in the framework.

13

MORPHEUS: A SUPPORTING TOOL FOR THE ATRIUM METHODOLOGY

> *«Perfection of means and confusion of goals seem, in my opinion, to characterize our age.»*
>
> — Albert Einstein
> German physicist and Nobel prize in Physics in 1921, 1879–1955

Software development process is always a challenging activity, especially because systems are becoming more and more complex. In this context, the MDD (Selic 2003) approach is gaining more and more attention from practitioners and academics. MDD has demonstrated positive influences for reliability and productivity of the software development process due to several reasons (Selic 2003): it allows one to focus on the ideas and not on the supporting technology; it facilitates not only the analysts get an improved comprehension of the problem to be solved but also the stakeholders obtain a better cooperation during the software development; etc. With those aims, MDD exploits models both to properly document the system and automatically or semi-automatically generate the final system. This is why the software development is shifting its attention (Bézivin 2004) from ''everything is an object'', so trendy in the eighties and nineties, to ''everything is a model''.

MORPHEUS is a tool which provides support for ATRIUM a methodology for the concurrent definition of requirements and software architectures.

ATRIUM (Navarro 2007; Montero and Navarro 2009) has been defined following the MDD principles, as models drive its application, and the tool MORPHEUS—see (Navarro 2011) for demos—has been built to support its models and activities. This methodology has been defined to guide the concurrent definition of requirements and software architecture, paying special attention to the traceability between them. In this context, the support of MORPHEUS is a valuable asset allowing the definition of the different models; maintaining traceability among them; supporting the necessary transformation, etc. This chapter focuses on MORPHEUS, its support to a MDD process, and how the MULTIPLE framework plays an important role on this tool. MORPHEUS was developed and funded in the context of the META[1] and MDDRehab[2] projects.

This chapter is structured as follows. After this introduction, a brief description of ATRIUM is presented in section 13.1. Section 13.2 describes the supporting tool of ATRIUM, MORPHEUS. Related works are described in section 13.3. Finally, section 13.4 ends this paper by presenting the conclusions and further works.

13.1 ATRIUM AT A GLANCE

ATRIUM provides the analyst with guidance, along an iterative process, from an initial set of user/system needs until the instantiation of the proto-architecture. ATRIUM entails three activities to be iterated over in order to define and refine different models and allow the analyst to reason about partial views of both requirements and architecture. Fig. 13.1 shows, using SPEM (OMG 2008c), the ATRIUM activities that are described as follows:

Modelling Requirements — This activity allows the analyst to identify and specify the requirements of the system-to-be by using the *ATRIUM Goal Model* (Navarro et al. 2006), which is based

1 *META: Models, Environments, Transformations and Applications*, ref. TIN2006-15175-C05. Department of Science and Technology (Spain) I+D+I.
2 *MDDRehab* project, ref. TC20091111. Universidad de Castilla–La Mancha.

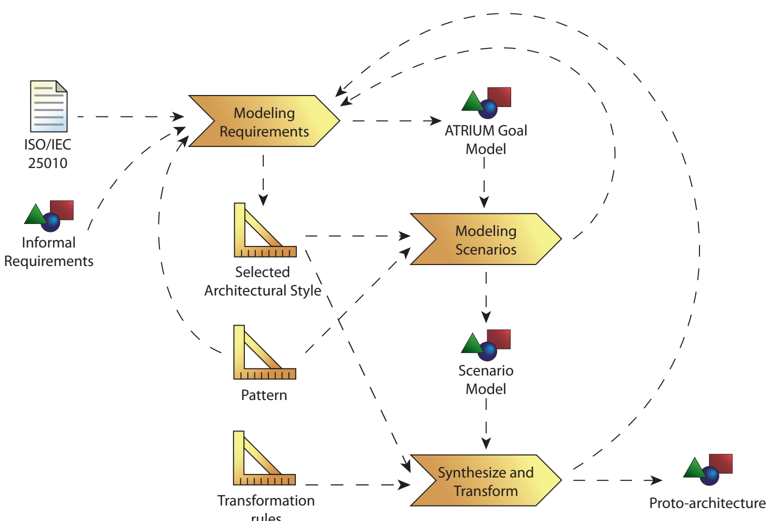


Figure 13.1: An outline of ATRIUM

on Knowledge Acquisition in autOmated Specification (KAOS) (Dardenne et al. 1993) and the Non-Functional Requirements (NFR) Framework (Chung et al. 2000). This activity uses as input both an informal description of the requirements stated by the stakeholders, and the CD 25010.2 Software product Quality Requirements and Evaluation (SQuaRE) quality model (ISO 2008). The latter is used as framework of concerns for the system-to-be. In addition, the architectural style to be applied is selected during this activity (Navarro 2007).

ATRIUM proposes an iterative process with three main activities: Modelling Requirements, Modelling Scenarios *and* Synthesize and Transform

Modelling Scenarios — This activity focuses on the specification of the *ATRIUM Scenario Model*, that is, the set of *Architectural Scenarios* that describe the system's behaviour under certain *operationalization* decisions (Navarro et al. 2007b). Each ATRIUM Scenario identifies the architectural and environmental elements that interact to satisfy specific requirements and their level of responsibility.

Synthesize and Transform — This activity has been defined to generate the proto-architecture of the specific system (Navarro and Cuesta 2008). It synthesizes the architectural elements from the ATRIUM scenario model that build up the system-to-be, along with its structure. This proto-architecture is a first draft of the final description of the system that can be refined in a later stage of the software development process. This activity has been defined by applying M2M transformation techniques (Czarnecki and Helsen 2006), specifically, using the QVT–Relations language (OMG 2008a) to define the necessary transformations. It must be pointed out that ATRIUM is independent of the architectural metamodel used to describe the proto-architecture, because the analyst only has to describe the needed transformations to instantiate the architectural metamodel he/she deems appropriate. Currently, the transformations to generate the proto-architecture instantiating the PRISMA architectural model (Pérez et al. 2006) have been defined. PRISMA was selected because a code compiler exists for this model.

ATRIUM and its supporting tool MORPHEUS have been validated using an industrial case study in the context of the EFTCoR project.

ATRIUM has been validated in the context of the tele-operated systems. Specifically, the Environmental Friendly and Cost-Effective Technology for Coating Removal (EFTCoR) (EFTCoR 2002-2005) project has been used for validation purposes. The main concern of this project was the development of a tele-operated platform for non-pollutant hull ship maintenance. In this chapter, we are going to use the specification made of the Robotic Device Control Unit (RDCU) to briefly illustrate how MORPHEUS provides support to each activity of ATRIUM. The RDCU is in charge of commanding and controlling in a coordinated way the positioning of devices along with the tools attached to them, however, it is not worth describing this case study in too much detail.

13.2 MORPHEUS: A MDD SUPPORTING TOOL

The main idea behind MORPHEUS is to facilitate a graphical environment for the description of the three models used by ATRIUM (*Goal Model*, *Scenarios Model*, and *PRISMA Model*) in order to provide the analysts with an improved legibility and comprehension. Several alternatives were evaluated such as the definition of profiles, or the use of meta-modelling tools. Eventually, we developed our own tool in order to provide the proper integration and traceability between the models.

Fig. 13.2 shows the main elements of MORPHEUS. The *Back-End* layer allows the analyst to access to the different environments, and to manage the projects he/she creates. Beneath this layer, the different environments of MORPHEUS are shown, providing each one of them support to a different activity of ATRIUM. The *Repository-Manager layer* is in charge of providing the different environments with access to the repository where the different models and metamodels are stored. In addition, each one of the graphical environments (*Requirements Model Editor*, *Scenario Editor*, and *Architecture Model Editor*) exploits *Microsoft Office Visio Drawing Control 2003* (Microsoft 2003)—*VisioOCX* in Figs. 13.3, 13.8 and 13.12—for graphical support. This control was selected to support the graphical modelling needs of MORPHEUS because it allows a straightforward management, both for using and modifying shapes. This feature is highly relevant for our purposes because all the kinds of concepts that are included in our metamodels can easily have different shapes, facilitating the legibility of the models. In addition, the user is provided with all the functionalities that *Visio* has, that is, she/he can manage different diagrams to properly organize the specification, make zoom to see more clearly details, print the active diagram, etc. In the following sections, each one of the identified environments is described.

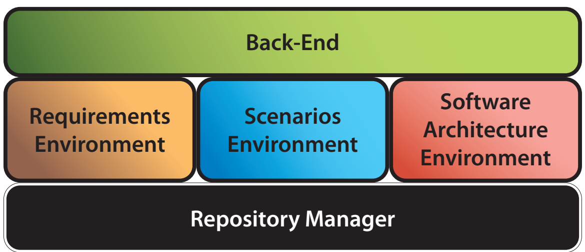


Figure 13.2: Main architecture of MORPHEUS

13.2.1 *Requirements Environment*

The Requirements Environment *is the subsystem of* MORPHEUS *in charge of supporting the* Modelling Requirements *activity.*

As described in section 13.1, *Modelling Requirements* is the first activity of ATRIUM. In order to support this activity, the *Requirements Environment* was developed. From the very beginning of the EFTCoR project, one of the main problem we faced was how the requirements metamodel had to change to be adapted to the specific needs of the project. With this aim, this environment was developed with two different work contexts. The first context is the Requirements Meta-Model Editor (RMME) in Fig. 13.3, which provides users with facilities for describing requirement meta-models customized according to project's semantic needs (see Fig. 13.4). The second context is the Requirements Model Editor (RME), also shown in Fig. 13.3, which automatically offers the user facilities to graphically specify models according to the active metamodel (see Fig. 13.7). These facilities are very useful to exploit MORPHEUS to support other proposals.

It can be observed in Fig. 13.4 that the RMME allows the user to describe new meta-elements by extending the core metamodel described in Fig. 13.5, that is, new types of artifacts, dependencies, and refinements. The applicability of this metamodel was was evaluated by analysing the existing proposals in requirements engineering (Navarro et al. 2006). For instance, Fig. 13.4 shows that the two meta-artifacts (*goal* and *requirement*) of the ATRIUM Goal Model were defined using the RMME. In order to fully describe the new meta-elements, the user can describe their meta-attributes and the

OCL constraints he/she needs to check any property he/she deems appropriate. Fig. 13.6 shows how the meta-artifact *goal* was defined by extending *artifact*; describing its meta-attributes *priority*, *author*, *stakeholder*, etc; and specifying two constraints (Fig. 13.6) to determine that the meta-attributes *stakeholder* and *author* cannot be null.

It is worth noting that automatic support is provided by the environment for the evolution of the model, that is, as the metamodel is modified, the model is updated in an automatic way to support those changes, asking the user to confirm the necessary actions whenever a delete operation is performed on meta-elements or meta-atributes. This characteristic is quite helpful because the requirement model can be evolved as the expressiveness needs of the project do.

Once the metamodel has been defined the user can exploit it in the modelling context, RME, shown in Fig. 13.7. It uses *VisioOCX* to provide graphical support, as Fig. 13.2 shows, and has been structured in three main areas. On the right side, the *stencils* allow the user to gain access to the active metamodel. Only by dragging and dropping these meta-elements on the drawing surface in the centre of the environment, the user can specify the requirements model. He/she can modify or delete these elements by clicking just as usual in other graphical environments. For instance, some of the identified goals and requirements of the EFTCoR are described in the centre of

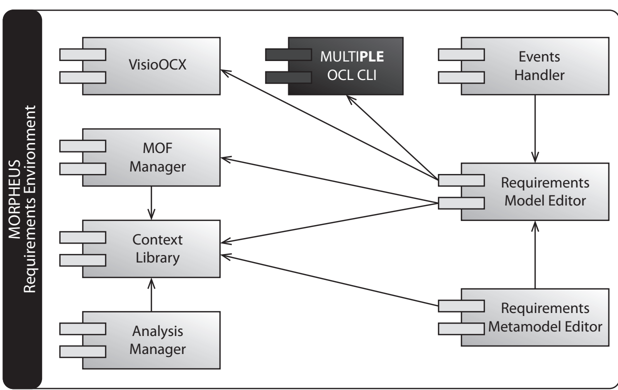


Figure 13.3: Main elements of the requirements environment

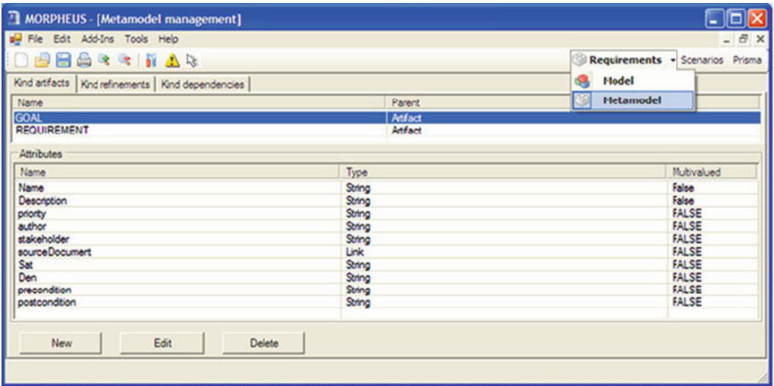


Figure 13.4: Meta-Modeling work context of the MORPHEUS Requirements Environment

the Fig. 13.7. On the left side of the RME, a browser allows the analyst to navigate throughout the model and modify it. As Fig. 13.2 illustrates, the *EventHandler* is in charge of manipulating the different events that arise when the user is working on the RME.

In addition, as Fig. 13.2 illustrates, the RME uses two components to provide support to OCL: MULTIPLE OCL-CLI and *MOFManager*. The former is an engine to check OCL constraints that was integrated in MORPHEUS. This OCL engine is provided by the MULTIPLE framework and corresponds with the *OCL Support CLI* component of the *Validation subsystem* (see section 8.5.2, page 203). The later was developed to allow us to manipulate metamodels and models in MOF (OMG 2006) format. This component behaves as a bridge be-

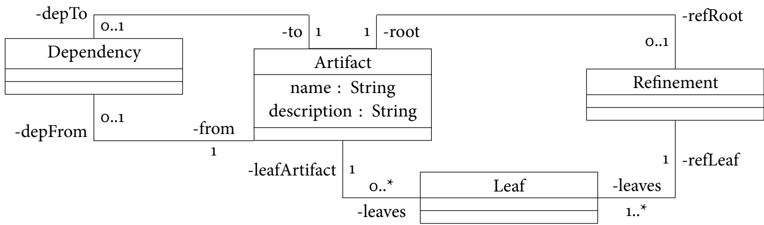


Figure 13.5: Core-metamodel for the requirements environment

tween EMF and MORPHEUS. By exploiting these components the constraints defined at the metamodel can be automatically checked. For instance, when the active diagram was checked, two inconsistencies were found that are shown at the bottom of the Fig. 13.7.

However, the support of the tool would be quite limited if it only provides graphical notation. For this reason, the *Analysis Manager*, shown in Fig. 13.2, has been developed to allow the user to describe and apply those rules necessary to analyse its models. These rules are defined by describing how the meta-attributes of the meta-artifacts are going to be valuated depending on the meta-atributes of the meta-artifacts they are related to by means of which meta-relationships. Once these rules are defined, the Analysis Manager exploits them by propagating the values from the leaves to the roots of the model (Navarro et al. 2007a). This feature can be used for several issues such as, satisfaction propagation (Navarro et al. 2007a), change propagation, or analysis of architectural alternatives (Navarro 2007).

13.2.2 *Scenario Environment*

As presented in section 13.1, *Modelling Scenarios* is the next activity of ATRIUM. This activity is in charge of describing the scenario mo-

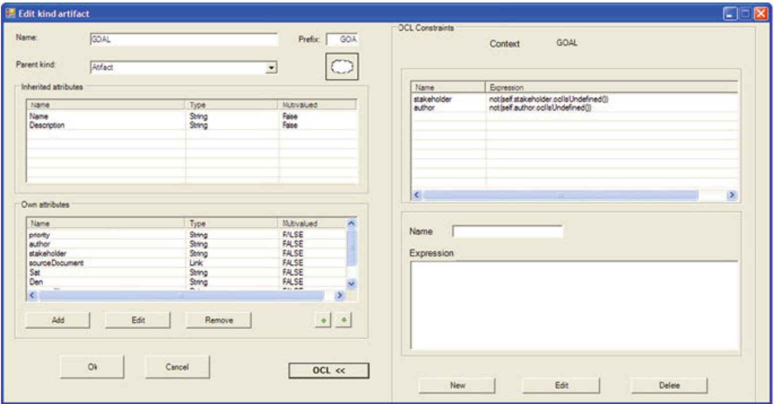


Figure 13.6: Describing a new meta-artifact in MORPHEUS

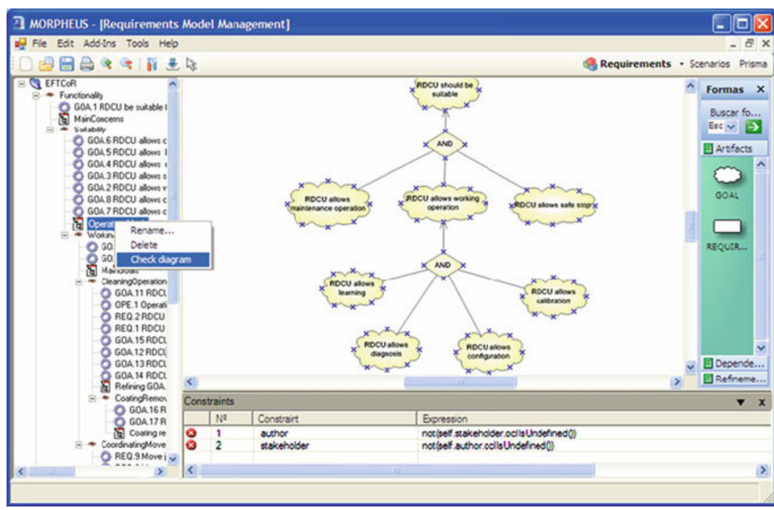


Figure 13.7: Modelling work context of the MORPHEUS Requirements Environment

The Scenarios Enviroment *is the subsystem of MORPHEUS in charge of supporting the* Modelling Scenarios *activitity. Scenarios are modelled using extended UML2 sequence diagrams.*

del. This model is exploited to realize the established requirements in the goal model by describing partial views of the architecture, where only shallow-components, shallow-connectors and shallow-systems are described. In order to describe these scenarios, an extension of the UML2 Sequence Diagram has been carried out to provide the necessary expressiveness for modelling these architectural elements (Navarro 2007). In order to provide support to this activity the Scenario Model Editor (SME), shown in Fig. 13.8, was developed. The *Scenarios Editor* uses the *VisioOCX* to provide the user with graphi-

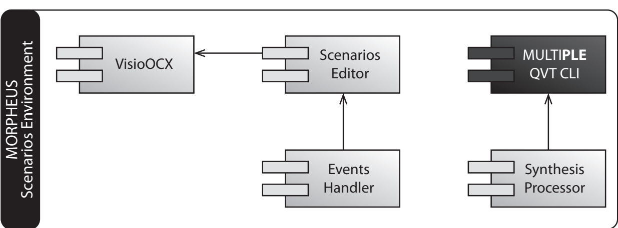


Figure 13.8: Main elements of the *Scenarios Environment*

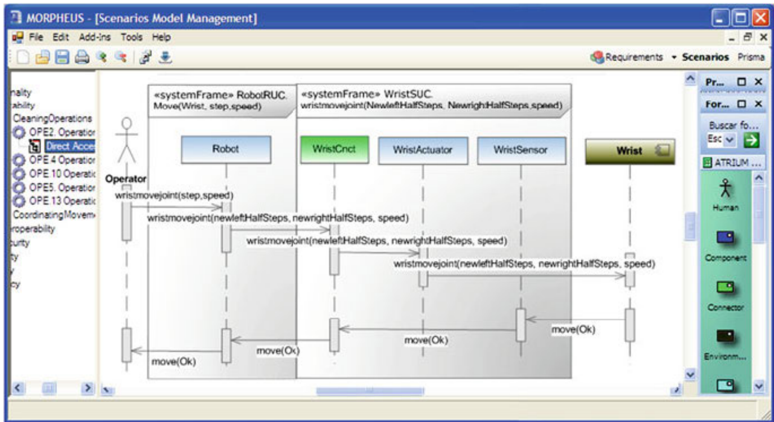


Figure 13.9: What the Scenario Model Editor looks like

cal support for modelling the Scenario Model. The *EventHandler* is in charge of managing all the events trigged by user actions. Fig. 13.9 illustrates how the SME has been designed. In a similar way to the RMME described in the previous section, it has been structured in three main areas. The *Model Explorer*, on the left, facilitates the navigation through the Scenario Model being defined in an easy an intuitive way and manages (creation, modification and deletion) the defined scenarios. It is pre-loaded with part of the information of the requirements model being defined. It facilitates to maintain the traceability between the Goal Model and the Scenario Model. In the middle of the environment is situated the *Graphical View* where the elements of the scenarios can be graphically specified. In this case, Fig. 13.9 depicts the scenario "OpenTool" that is realizing one of the operationalizations of the goal model. It can be observed how several architectural and environmental elements are collaborating by means of a sequence of messages. On the right side it can be seen the *Stencil* that makes available the different shapes to graphically describe the ATRIUM scenarios.

Another component of the Scenario Environment is the *Synthesis processor* (see Fig. 13.10). It provides support to the third activity of ATRIUM *Synthesis and Transform* which is in charge of the genera-

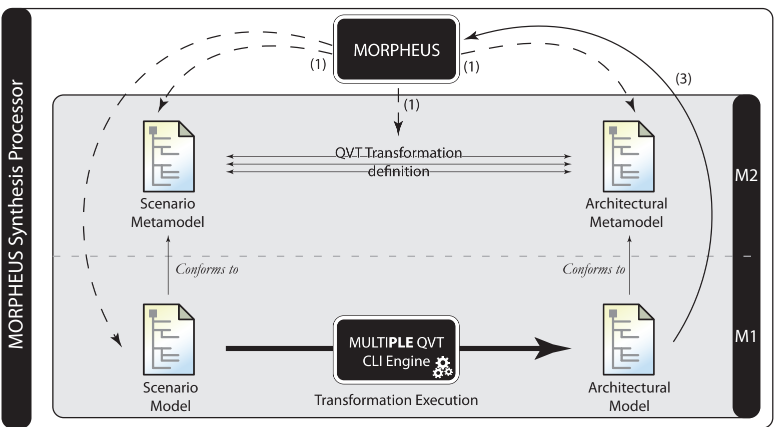


Figure 13.10: Describing the Synthesis processor

The Synthesis processor *makes use of the* MULTIPLE QVT *command-line interface.*

tion of the proto-architecture. For its development, the alternative selected was the integration an existing M2M transformation tool. The features that the candidates tools had to provide were, first, to support the QVT–Relations language and second, to be easily integrable with the existing work. Specifically, the MULTIPLE QVT-CLI

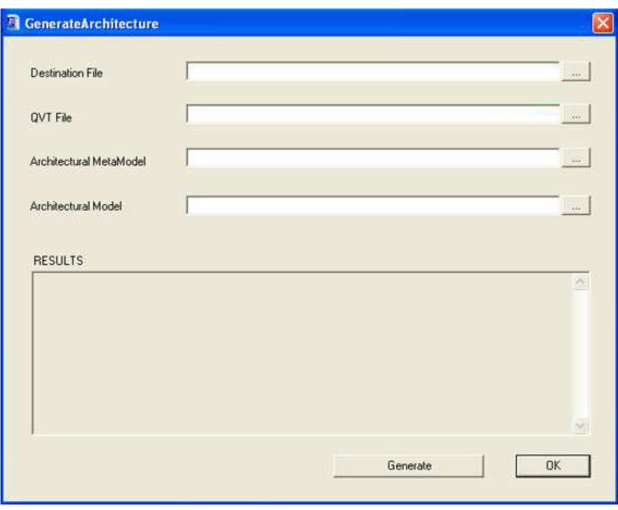


Figure 13.11: Generate architecture dialog

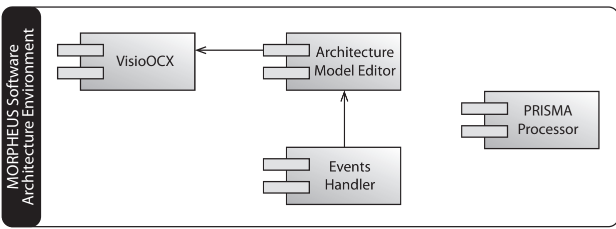


Figure 13.12: Main elements of the Software Architecture Environment

tool was chosen as Fig. 13.12 illustrates. It accepts as inputs the metamodels and their corresponding models in XMI format to perform the transformation. This engine is invoked by the *Synthesis processor* which proceeds in several steps. First, it stores the Scenario Model being defined in XMI. Second, it provides the user with a graphical control to select the destination target architectural model, the QVT transformation to be used and the name of the proto-architecture to be generated. By default, PRISMA is the selected target architectural model because the QVT rules (Navarro 2007) for its generation have been defined. However, the user can define its own rules and architectural metamodels to synthesize the Scenario Model. Finally, the *Synthesis processor* performs the transformation by invoking the QVT engine. The result is an XMI file describing the proto-architecture. Fig. 13.11 shows the dialog which provides the UI to invoke the synthesis processor with the corresponding fields to select each one of the source and target files.

13.2.3 *Software Architecture Environment*

As can be observed, both the Requirements Environment and the Scenario Environment provide support to the three activities of ATRIUM. However, as specified in section 13.1, a proto-architecture is obtained at the end of its application. This proto-architecture should be refined in a latter stage of development to provide a whole description of the system-to-be. With this aim the *Software Architecture Environment* (Perez et al. 2006) was developed. It makes available

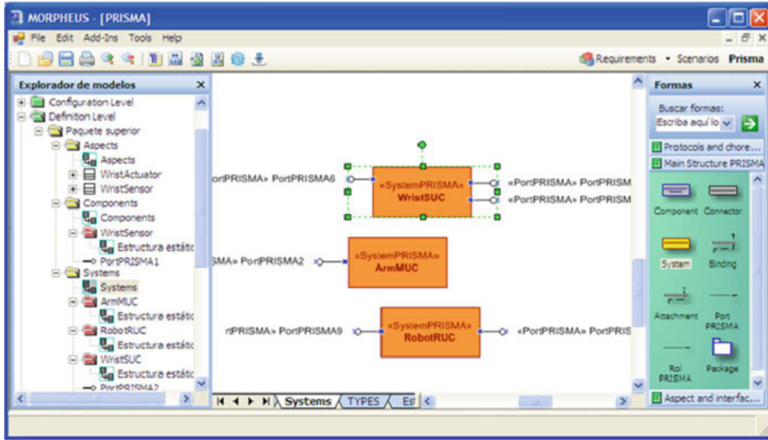


Figure 13.13: What the Architectural Editor looks like

a whole graphical environment for the PRISMA-ADL (Pérez et al. 2006) so that the proto-architecture obtained from the scenarios model can be refined.

The Software Architecture Environment *provides full support for the PRISMA Architecture Description Language, allowing the use of the PRISMA-MODEL-COMPILER.*

As Fig. 13.13 depicts, this environment integrates *VisioOCX* for graphical support in a similar way to the previous ones. The *Architectural Model Editor* is the component that provides the graphical support, whose appearance can be seen in Fig. 13.12. It has three main areas: the stencil on the right where the PRISMA concepts are available to the user, the graphical view in the centre where the different architectural elements are described; and the model explorer on the right. It is worthy of note that this browser is structured in two levels following the recommendation of the ADL (Perez et al. 2006): definition level, where the PRISMA types are defined; and configuration level where the software architecture is configured.

As this environment should allow the user to refine the proto-architecture obtained from the synthesis of the scenario model, it provides her/him with facilities to load the generated proto-architecture if PRISMA was the selected target architectural model. In addition, it also provides an add-in that facilitates the generation of a textual

PRISMA specification, which can be used to generate C# code by using the PRISMA framework.

13.3 RELATED WORKS

Nowadays, MDD is an approach that is gaining more and more followers in the software development area, and lots of tools that support this trend have arisen. Nevertheless, none of the existing solutions can completely cover the capabilities of the MORPHEUS tool.

The Eclipse Modeling Framework (EMF) has become one of the most used frameworks to develop model-based applications. EMF provides a metamodelling language, called *Ecore*, that can be seen as an implementation of the EMOF language. Around EMF lots of related projects have grown that complement its modelling and metamodelling capabilities, such as OCL interpreters, model transformation engines, or even tools able to automatically generate graphical editors, such as GMF (Eclipse 2011e). The advantages are twofold: first they are usually quite mature tools, and second it is easy to interoperate with them by means of the XMI format. That is why the MORPHEUS tool has the *MOFManager* component: it allows us to reuse these tools as is the case of the MULTIPLE OCL checker and the MULTIPLE model transformations engine. Nevertheless, a solution completely based in EMF has also some important drawbacks. The main one is that, although it is not mandatory, this framework and its associated tools are fundamentally designed to deal with static models that do not change at run time. This factor makes frameworks like GMF completely useless for our purposes, because in MORPHEUS the requirements metamodel is populated with instances during its evolution and it is necessary to be able to synchronize them.

As opposed to other pure MULTIPLE-based tools, MORPHEUS does not use GMF to implement the different graphical editors it provides. This is due to the inability of GMF to easily deal with models that evolve at run time.

Other analyzed alternatives are the *MS DSL Tools* (Cook et al. 2007). *MS DSL Tools* are a powerful workbench that also provides modelling and metamodelling capabilities to automatically generate both code and graphical editors in Visual Studio. However, it exhibits the same weakness than the previous solution: it is basically designed to deal with models that do not evolve during time, so that,

these models can only be modified during design time and not at run time. Moreover, it lacks of the wide community that provides complementary tools to deal, check and analyze models, in comparison with the solution that is completely based on EMF. This disadvantage is also present in other tools, like the ones associated to meta-CASE (Gong et al. 1997) and domain specific modelling techniques, such as MetaEdit+ (Metacase 2007; Kelly et al. 1996).

13.4 CONCLUSIONS AND FURTHER WORKS

In this chapter a tool called MORPHEUS has been presented paying special attention to how it provides support to a MDD process, ATRIUM, and how this tool makes use of the functionality provided by MULTIPLE to enrich this MDD process. It has been shown how each model can be described by using this tool and, specially, how traceability throughout its application is properly maintained. It is also worth noting the meta-modelling capabilities it has, providing automatic support to evolve the model as the metamodel is changed. The integration of an OCL checker is interesting as it allows the user to evaluate the model using the properties he/she deems appropriate. The use of QVT–Relations to generate the architectural models is another key point. This declarative language enables the use of any architectural model that fits the user's needs.

Part VI

CLOSURE

SUMMARY

This part closes this thesis. Chapter 14 describes some relevant works which are closely related to the MULTIPLE framework and proposal. Specifically, this chapter focuses in related works in the areas of feature modeling and SPLs. Next, chapter 15 summarizes the contents of this thesis and presents the conclusions of the work. Finally, 16 describes the most relevant works, which have been produced throughout the development of this thesis, that have been published in different journals and conferences, both national and international.

14 RELATED WORKS

«Competition is not only the basis of protection to the consumer, but is the incentive to progress.»

— Herbert Clark Hoover
31st (1929–1933) President of the United States, 1874–1964

Model Driven Engineering, Feature Modeling and Software Product Lines have been an important discussion topic in the Software Engineering community. There are many studies on these subjects and a great amount of proposals have arisen.

This chapter summarizes other works that are closely related with the MULTIPLE framework and the proposal described in this thesis. Since our work covers several paradigms and stages of the development of a SPL, related works can be grouped in the following topics: MULTIPLE and other feature modeling proposals (including the upcoming variability management proposal from the OMG); feature models and class diagrams; constraints in feature modeling and; MULTIPLE and other SPLE approaches (both classical and model-based). Next, they are presented.

14.1 MULTIPLE FEATURE MODELS AND OTHER FEATURE MODELING PROPOSALS

Most of the proposals for feature modeling are based in the original FODA notation. Such proposals have contributed several extensions to it (Chen et al. 2009). Our work is closely related with previous research in feature modeling, however, there are several distinctive aspects:

Our proposal for feature modeling is strongly based on previous literature. However MULTIPLE *stresses in a key distinctive aspect: configurations should be considered as actual instances of feature models to allow their use in complex* MDE *processes.*

Czarnecki and Kim (2005) propose a notation for cardinality-based feature modeling. In this sense, our tool shares most of this notation as it is widely known and used, but we have included some variants. First, in our approach features can not have an *attribute type*, but rather, they can have typed *feature attributes* which can be used to describe *parameterized features*. Second, according to Czarnecki and Kim *op. cit.* both feature groups and grouped features can have cardinalities. However, the possible values for grouped features cardinalities are restricted. In our proposal, these values are not restricted and have different meanings: cardinality of feature groups specify the number of features that can be instantiated, and cardinality of grouped features specify the number of instances that each feature can have.

Our work describes a prototype to define configurations of feature models. Previous work has been also done in this area, such as the Feature Modeling Plugin (FMP) (Antkiewicz and Czarnecki 2004). This tool allows the user to define and refine a feature model and configurations by means of specializations. The advantage of this approach is that it is possible to guide the configuration process by means of constraint propagation techniques. The main difference with our work is that in FMP configurations are defined in terms of the feature metamodel and both models and configurations coexist at the same layer. Thus, in order to be able to deal both with models and configurations it is necessary to build complex editors (as they must guarantee that the specialization process—as explained in chapter 6—is properly done).

Besides the difference stated before (configuration by specialization *vs.* configuration by instantiation), our tool also provide a more intuitive graphical metaphor. We state that because feature models in MULTIPLE are represented using the traditional notation (i. e., trees of boxed features linked with decorated lines). This advantage of MULTIPLE with regard to other tools not only applies for FMP, but also for other feature modeling tools such as PURE::VARIANTS (Beuche 2007) or FEATUREMAPPING (Heidenreich et al. 2008). An example of such advantage is that MULTIPLE is used by external researchers which are not related with this thesis and its case studies, such as Duran-Limon et al. (2011);

14.2 MULTIPLE AND THE OMG COMMON VARIABILITY LANGUAGE

Common Variability Language (CVL)—(OMG 2009)—is the OMG's upcoming standard for variability management. At this moment, CVL is still under development and only the *Request for proposal (RFP)* document is publicly available. However, with respect to the RFP, we can state that MULTIPLE is strongly aligned with the upcoming CVL standard, since: (i) MULTIPLE uses other MOF standards (MOF, UML, OCL, QVT, XMI); (ii) MULTIPLE is a functional tool based on EMF, which guarantees interoperability; (iii) MULTIPLE can express variability on models using the most common variability mechanisms; (iv) MULTIPLE provides support for defining and checking complex constraints; (v) in MULTIPLE, variability is defined as a separate model and semantics of the variability language are defined by using QVT; etc. Specifically, MULTIPLE complies with sections 5.1.x, 5.2.5, 6.1, 6.4.1, 6.5.1.x and 6.5.2.x of the CVL RFP (op. cit.).

The MULTIPLE *proposal for variability management is strongly aligned with the upcoming Common Variability Language* MOF *standard.*

14.3 FEATURE MODELS AND CLASS DIAGRAMS

Some previous works have already represented feature models as class diagrams. In Czarnecki and Kim (2005) the translation from

feature models to class models is performed manually, and no set of transformation rules are described. In this work, OCL is also presented as a suitable approach to define model constraints, but as the correspondences between feature models and class diagrams are not precisely defined, there is no automatic generation of OCL invariants.

Several proposal which represent feature models using class diagrams have arisen in the last years. However, none of them cover all the same aspects than MULTIPLE *does: i. e., automatic generation of class diagrams and model constraints by using model transformations and automatic support for model configurations definition.*

Laguna et al. (2008) do present a set of QVT rules to automatically generate class diagrams from feature models. However, in this case, neither model constraints nor configuration definitions support is presented.

14.4 FEATURE MODEL CONSTRAINTS

Batory (2005) present a proposal for feature constraints definition and checking. Specifically, this work proposes to represent features as propositions and restrictions among them are represented as propositional formulas. However, in propositional formulas only `true` and `false` values are allowed. This approach is not suitable to our work, as we can have typed attributes which can not be expressed by this kind of formulas. Thus, we state that more expressive languages are needed. In this case, we propose OCL as our constraint definition language.

To use DVMs allows us to address some satisfiability problems from new points of view. The introduction of cardinalities and unbounded attribute types makes harder to reason about feature models (for example, satisfiability of feature models). Thus, richer formalisms (compared with the traditional ones) are needed.

The FAMA framework, which was presented in previous chapters, has advanced in this are, and not only propositional formulas are used to reason about feature models and their configurations, but more powerful formalisms are integrated in this framework. The versatility of the FAMA framework has been a key point to choose such a framework to perform model-checking tasks in MULTIPLE. However, FAMA is still unable to deal with every constraint that can be defined in MULTIPLE, since out tool provides FMCL an OCL-based

language to define model constraints which is can not be translated to FAMA specifications.

Current approaches for class diagrams model checking can be used to validate cardinality-based feature models, going beyond the current capabilities of the state of the art proposals for feature models' verification.

Fortunately, class diagrams and OCL are widely used and known, and several formalisms to reason about them have been proposed. In this sense, some interesting works have been published in this are, and we have already done some preliminary works in model consistency checking by using third party formal tools. Specifically, we have reused UML2CSP (Cabot et al. 2007; Cabot et al. 2008) a formal framework which is able to transform class diagrams, plus the OCL constraints that they contain, to ECL^{i}PS^{e} (Apt and Wallace 2007), a variant of *Prolog* (Sterling and Shapiro 1986) for constraint programming.

14.5 FEATURE MODELS AND OTHER SPL APPROACHES

Batory et al. (2006) capture the domain features in a feature model. In BOM, the case study where MULTIPLE is demonstrated, we capture features in two kinds of feature models. In our research, we observed that the variability problem is not solved by means of a unique feature model and the monotonic gluing of these features. We have taken a new approach, which manages the variability in two phases (one by building a base architecture using the domain features, and another one by decorating these base architectures with the application domain features) in order to obtain the final product.

Trujillo (2007) uses Feature Oriented Programming (FOP) as a technique for inserting features into XML documents by means of XSLT templates. In BOM-Lazy we use this technique but at the model level—i. e. we use Feature Oriented Modeling (FOM)—by means of QVT–Relations Transformations. The features are inserted on the skeleton model in order to obtain the PRISMA architecture model.

Clements and Northrop (2001) use the SPL development approach considering a clear division between domain engineering and application engineering phases for the reuse and the automation of the software process. In BOM–Lazy, we have already used this approach to develop our SPL.

Avila-García et al. (2006) use process modeling in SPEM to pack reusable assets. In our approach we use this OMG standard, in conjunction with some other standards, when we model the production plan.

Bachmann et al. (2004) propose to separate the variability declaration of the affected assets in separate artifacts. In BOM, the specification of the variability and the functionality are captured in different feature models. The use of instances of such feature models allows the user to input the information of the domain features. Those features allow to build the associated assets, or to define the application domain features in order to configure the final application.

The AMPLE project *is a reference initiative in the aspect-oriented model-driven* SPLs *field. It has been developed by six research centres and three industrial organisations.*

Regarding Model Driven Software Product Line Engineering (MDSPLE) approaches, the *AMPLE project* (AMPLE 2011) emerges as a reference initiative in the Aspect-Oriented Model-Driven Software Product Lines (AO-MD-SPL) field. This project was developed by a consortium of six research centres in the areas of SPLs, AOSDs and MDE and three industrial organisations working with or seeking to deploy product line solutions (Rashid et al. 2011). Although the AMPLE project is focused in AOSD it is strongly rooted in MDE, and thus, it is comparable in some aspects with MULTIPLE.

The AMPLE project covers all the development stages of a SPL, from the requirements elicitation (Sardinha et al. 2009; Weston et al. 2009; Shakil Khan et al. 2008) to the code generation stage (Fuentes et al. 2009; Groher and Voelter 2009), where the final product is obtained.

Groher and Voelter (2009) present a summary of the AMPLE proposal where PURE::VARIANTS is used to define feature models, which are then transformed to an *equivalent* EMF-based custom metamodel. According to this work, an AO-MD-SPL process must be structured in 3 stages: *problem space modeling* (domain model), *solution space modeling* (PIM) and *solution space implementation* (PSM, code). Models are transformed from a stage to the next one using model transformations. This staged process resembles the staged production plan proposed in the application of MULTIPLE to the BOM–Lazy case study. However, some differences arise:

First, variability is managed in a different way. In BOM–Lazy there is not a separation between problem space modeling and solution space modeling. In BOM we always deal with domain concepts (whether they are *domain features* or *application domain features*). In BOM, solution space concepts are represented by using modular models, component–connector models or PRISMA models. Moreover, the third stage (the implementation stage) is fully automatic in BOM, thanks to the PRISMA-MODEL-COMPILER.

Second, variability is managed in AMPLE by using aspects. Features usually impact in the functionality of different modules or components, in the same way than cross-cutting concerns impact in different artifact in Aspect-Oriented Programming (AOP) (Filman et al. 2005). This way, using aspects to describe features is a effective and straightforward approach. In MULTIPLE, and specifically in the BOM case study, we have modeled such impact from a generic point of view by using a set of QVT rules which describe architectural patterns. Using such patterns in a transformations engine we can transform the affected artifacts (for example, by adding services, roles, ports, etc.). Nevertheless, an AOSD approach can also be implemented in MULTIPLE by using the PRISMA metamodel, which is a language to describe aspect-oriented software architectures.

Third, although AMPLE is a model-based approach, feature model configurations are not expressed as instances of their corresponding feature models. This issue has a big impact in the AMPLE philosophy, leading to the definition of several variability management languages which provide support for model transformations in different stages of a SPL process. Such variability management languages, which usually are textual languages, are in charge of relating feature model configurations and different structural models (problem space models, solution space models, aspect models, etc.).

Specifically, Groher and Voelter (2009) use a family of different languages to adapt the different models based on the feature selection, i. e. *XWeave*, *XVar*, *Xtend* and *Xpand*. Each one of these languages are designed with a specific purpose, which has the advantage of being compact, and easy to learn and understand. However, they

also have their own syntax and limitations: some can only deal with models conforming to the same metamodel (e. g. *XWeave*) while others can only be used to define mappings to filter structural models (e. g., *XVar*). In MULTIPLE only one language is required (QVT), which is able to deal with models conforming any metamodel, and can be used to define any kind of pattern, filter or expression. Moreover, QVT is a high level declarative language which can be graphically represented and easily understood. For example, Groher and Voelter (2009) use *Xpand* to define aspect templates. In BOM a similar task is performed by using the *T2* transformation (see section 9.5.2, page 260), which populates a skeleton architecture using the *application domain features*.

In contrast to the MULTIPLE approach, where only one language to define the mappings between models and configurations is needed, Zschaler et al. (2010) propose *VML**, a product line to generate variability management languages. Such a proposal aims to help SPL designers to define the mappings between configurations and structural models (since the lack of a proper representation for feature model configurations can be problematic, avoiding the use of high level declarative languages). This proposal is based on the premise that general-purpose model transformation languages place too heavy a burden on SPL engineers. *VML** provides the methods and tools to define DSLs for variability management. Specifications in such DSLs are in the end translated to a general purpose model transformation language, hiding the intricacies of MDE to SPL designers.

15 SUMMARY AND CONCLUSIONS

> *«In nature there are no rewards or punishments, there are only consequences.»*
>
> — Horace Annesley Vachell
> English writer, 1861–1955

In this thesis we have presented MULTIPLE, an approach to effectively represent and manage variability in complex MDE processes. This work starts from the current feature-based approaches to variability management and tries to solve some of their drawbacks. Large software systems like the Linux kernel feature model—5400 features approx. (She et al. 2010)—or the Rolls-Royce feature model—1200 features approx. as shown in chapter 10—serve as an example about how feature models are not fully exploited, i. e., it is uncommon the use of cardinality-based feature models; and feature models hardly contain typed feature attributes. Such poorly-defined models lead us to forget that feature models configurations are, in fact, instances of feature models. Such idea takes us to avoiding the use of feature models as active parts in MDE processes.

The production plan of a SPL is the set of steps that must be done to obtain a product given a specific configuration and a set of core assets. However, in traditional SPLs this process is reduced to a gluing of code snippets because MDE techniques are not used. However,

these techniques raise the level of abstraction and provide more expressiveness to describe software artifacts. This provides independence from the implementation platform, and allows to automate several tasks. Moreover, MDE is becoming a mature discipline which provides a wide range of standards and tools which support this trend.

MDPLE emerges as a first approach to integrate both MDE and SPLs. This proposal raises the level of abstraction, as the main assets of a SPL are models, instead of code fragments. However, the same limitations about variability management still remain. This is because variability is not considered as an active model which describes a new view of the systems to be.

The use of multi-models allows to use different DSLs in the development of software systems.

The use of multi-models allows describing a system using the DSL that the developer deems appropriate. The use of different modeling languages reduces the learning curve and allows to integrate different technical spaces. It also eases the decomposition of a system in different views which simplify the development process. Moreover, the use of multi-models allow to integrate modern software development paradigms.

Given this situation, we have shown how the multi-modeling techniques can ease the development of SPLs, designing what we describe as Multi-Model Driven Software Product Lines (MMDSPLs). In a MMDSPL the system's variability view is managed in an explicit way, using a dedicated model and specific notation. This way a variability model is an active asset in the software development process. Moreover, in a MMDSPL model transformations can (and should) be used in any part of the development process. The key elements which enable the use of variability models using current modeling tools are the DVMs. A DVM is an equivalent and intermediate artifact which allows the definition of feature model configurations.

This thesis provides the following contributions:

FIRST, it extends previous works about software system views. We have adapted the concept of system view to the multi-modeling approach and techniques. Moreover, we provide an operationalization of Limón Cordero's proposal providing an implementation

its metamodels (specifically, modular and component-connector metamodels), and the corresponding graphical DSLs.

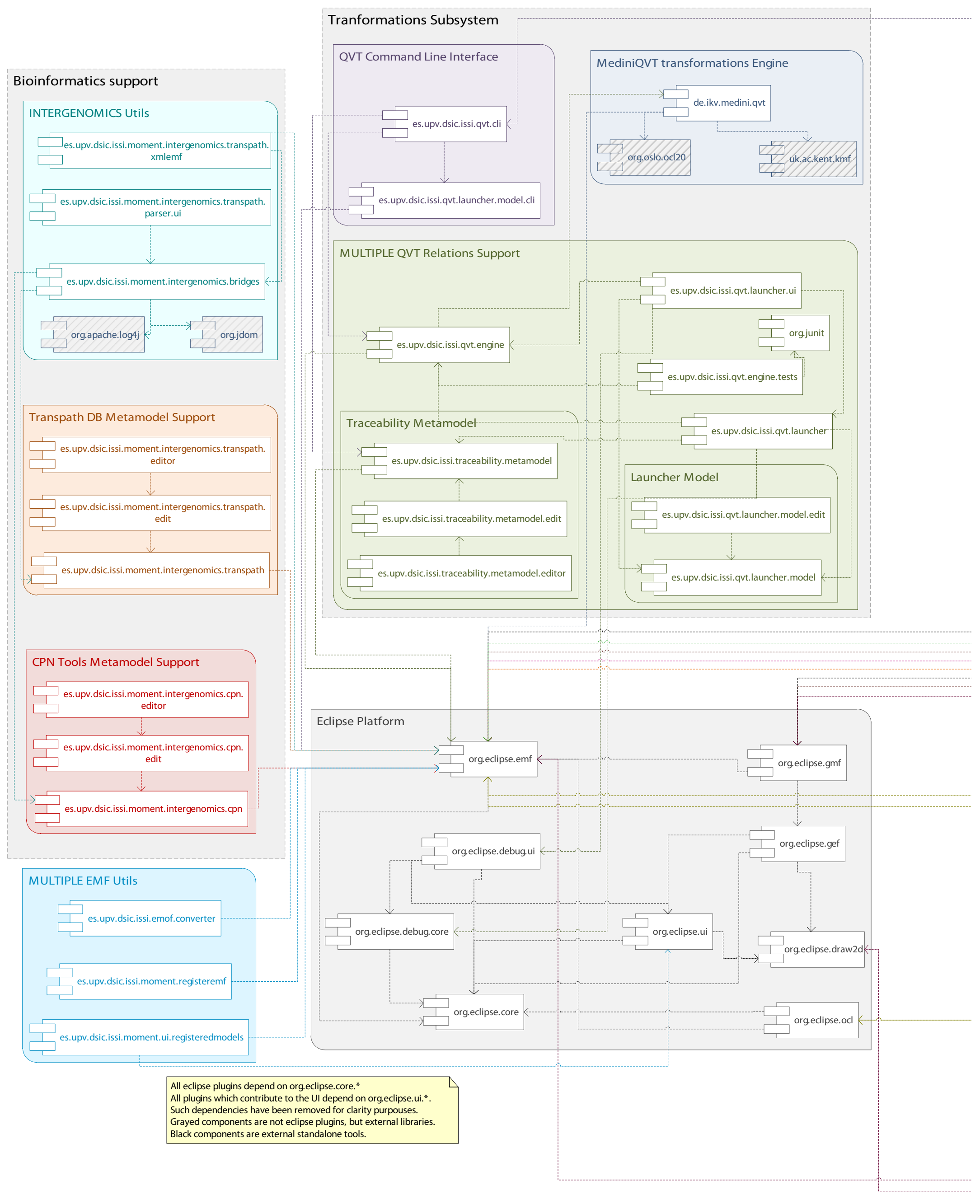
SECOND, this thesis introduces variability modeling as an additional view of the system. Specifically, this work puts feature models and feature model configurations in the context of MOF. This implies that each software artifact is precisely defined at its corresponding level of abstraction.

THIRD, we propose a MOF-compliant metamodel to describe system's variability. This metamodel is very expressive, as it can be considered as a superset of the most important contributions in the variability management field over the last 20 years. Furthermore, the proposal allows to overcome the main issues which arise when trying to use feature models and feature model configurations in recent metamodeling tools. The use of the so called DVMs allows maintaining the *instance-of* relationship between feature models and configurations. This relationship guarantees the consistency between feature models and configurations for free, which turns out to be more simple. Moreover, we provide a constraint description language (FMCL) to describe complex model constraints; and the semantics of our metamodel and the constraint description language are clearly defined by a set of equivalence relationships among them and MOF and OCL.

This thesis proposes to manage variability in an explicit way in a MOF compliant framework, taking advantage of existing standards and tools.

Thanks to the equivalence relationships, feature models can be instantiated by using the DVMs; and such models and instances can be validated using common tools such as OCL checkers. Furthermore, the use of class diagrams to describe variability models enables new paradigms for variability model checking, using for example the tools to check UML2 class diagrams.

FOURTH, we provide an implementation of our proposal as an integrated framework for MMDPLE. The framework, called MULTIPLE, is a tool which stands out for its genericity, interoperability, extensibility and ease of use. Fig. 15.1 shows the full architecture of the MULTIPLE framework including all the additions that have been contributed to the initial framework throughout the development of the different case studies. MULTIPLE provides support for describing the


Tranformations Subsystem
QVT Command Line Interface
es.upv.dsic.issi.qvt.cli
es.upv.dsic.issi.qvt.launcher.model.cli
MediniQVT transformations Engine
de.ikv.medini.qvt
org.oslo.ocl20
uk.ac.kent.kmf
MULTIPLE QVT Relations Support
es.upv.dsic.issi.qvt.launcher.ui
es.upv.dsic.issi.qvt.engine
org.junit
es.upv.dsic.issi.qvt.engine.tests
Traceability Metamodel
es.upv.dsic.issi.qvt.launcher
es.upv.dsic.issi.traceability.metamodel
es.upv.dsic.issi.traceability.metamodel.edit
es.upv.dsic.issi.traceability.metamodel.editor
Launcher Model
es.upv.dsic.issi.qvt.launcher.model.edit
es.upv.dsic.issi.qvt.launcher.model
Bioinformatics support
INTERGENOMICS Utils
es.upv.dsic.issi.moment.intergenomics.transpath.xmlemf
es.upv.dsic.issi.moment.intergenomics.transpath.parser.ui
es.upv.dsic.issi.moment.intergenomics.bridges
org.apache.log4j
org.jdom
Transpath DB Metamodel Support
es.upv.dsic.issi.moment.intergenomics.transpath.editor
es.upv.dsic.issi.moment.intergenomics.transpath.edit
es.upv.dsic.issi.moment.intergenomics.transpath
CPN Tools Metamodel Support
es.upv.dsic.issi.moment.intergenomics.cpn.editor
es.upv.dsic.issi.moment.intergenomics.cpn.edit
es.upv.dsic.issi.moment.intergenomics.cpn
Eclipse Platform
org.eclipse.emf
org.eclipse.gmf
org.eclipse.gef
org.eclipse.debug.ui
org.eclipse.debug.core
org.eclipse.ui
org.eclipse.draw2d
org.eclipse.core
org.eclipse.ocl
MULTIPLE EMF Utils
es.upv.dsic.issi.emof.converter
es.upv.dsic.issi.moment.registeremf
es.upv.dsic.issi.moment.ui.registeredmodels
All eclipse plugins depend on org.eclipse.core.*
All plugins which contribute to the UI depend on org.eclipse.ui.*.
Such dependencies have been removed for clarity purpouses.
Grayed components are not eclipse plugins, but external libraries.
Black components are external standalone tools.

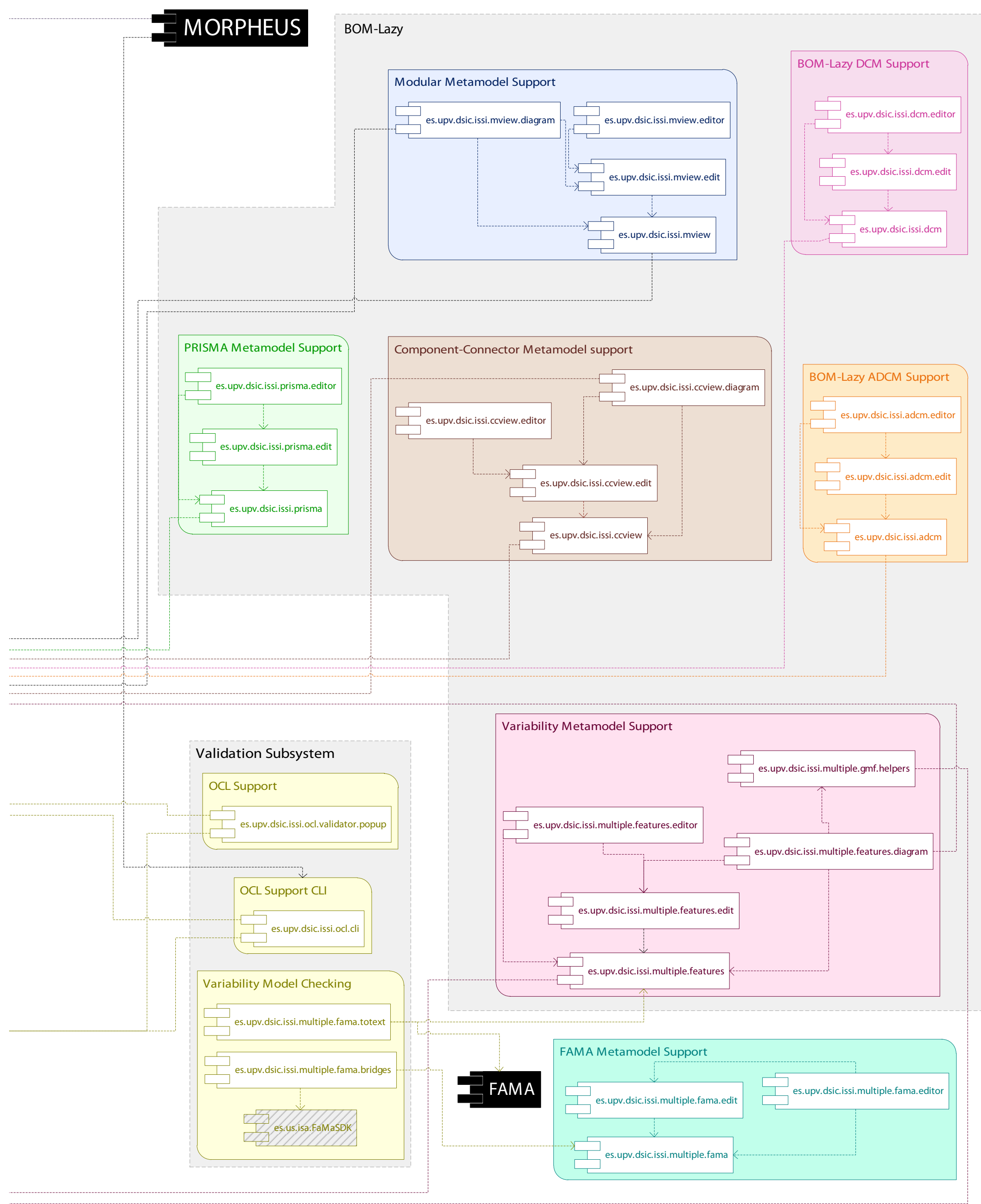


Figure 15.1: Extended architecture of the MULTIPLE framework

variability view, the modular view and the component–connector view using the corresponding graphical notations. Additionally, it provides support to deal with FAMA feature models and PRISMA architectural models. The tool integrates a model transformations engine, which can be accessed using the graphical UI or using the command-line; and the same applies for the integrated OCL engine. The tool also integrates the FAMA model-checker which allows to analyse feature models. Finally, a set of model transformations are provided within the framework (e. g., *Feature2ClassDiagram* or *MultipleFeatures2FamaFeatures* among others, see appendixes A and D).

FIFTH, proof of these features of the MULTIPLE framework is that it has been successfully applied in five different case studies of the most diverse nature.

The MULTIPLE *has been validated in five different case studies and the feature modeling editor is used by external researches.*

The MULTIPLE framework and the proposed methodology has been used to modernize the BOM–Eager SPL. As a result the BOM–Lazy proposal has been designed. From this case study we can conclude that an explicit variability management allows to concentrate all the efforts in early stages of a SPL development. This way, all the work is done in the domain engineering phase; and it is in this phase when the domain engineer captures all the knowledge required to generate the final product. Moreover, the use of feature models and other system's views allows capturing such knowledge in an explicit way using QVT to describe the relationships among the different views (i. e., a model transformation). In this case it is noteworthy that a feature model is an active asset that can be used and reused directly by means of MDE techniques. Besides, to define a model transformation to generate the final products allows us to include additional information in such transformations, such as well design patterns. This opens new paths to analyse quality aspects in SPLs, as in the case of a SPL the quality of the final product is not only related with the quality of the core-assets but also with the quality of the production plan—the transformation—itself (González-Huerta 2010; González-Huerta 2011).

Furthermore, the use of explicit and instantiable feature models and the use of model transformations provide more flexibility, as the

software architectures are defined by means of patterns at model level. Moreover, the use of software patterns provide greater scalability, as the relationships among the system's views are defined regardless of the number of products of the SPL. In this case, the different products of the SPL are defined in an implicit way by the QVT transformation. Finally, in the case of BOM–Lazy the implementation efforts are reduced drastically thanks to PRISMA and the PRISMA-MODEL-COMPILER.

It is necessary to provide validation and verification mechanisms in industrial environments. Large scale feature models are very complex and error-prone, and consequently, verification and validation is not straightforward and can not be performed by hand.

We can also conclude that it is required to provide the necessary mechanisms to validate, check and analyse feature models and their constraints. In the study of the industrial feature model we found that the majority of the errors were introduced by abusing the notation. This abuse is produced by a poorly implemented feature modeling tool. A feature modeling tool must provide the necessary mechanisms to ensure that the feature models are not invalid. Having an invalid feature model in a context where it is used for documentation purposes is just problematic; but to have such a model in a MDE process can not be admitted, as the feature model is a fundamental asset on top of which the process is built. However, it is not only necessary to ensure that the notation is properly used, but also the model checking mechanisms are fundamental. Proof of this is that in the industrial case study we found that the feature model was void, and it did not represent any product at all. Nevertheless, more research in such issues is still required, as some scalability problems arise when using current tools. In this sense, the relationships which add more complexity are the cross-tree ones.

We can confirm the benefits of applying the MULTIPLE framework to different case studies. Such case studies can be, in the end, considered as simple MMDPLE processes where the system variability view is trivial (i. e., only a single product). We can verify that the solutions based on MDE and MULTIPLE are more interoperable and allow us to deal with heterogeneous data sources (such as in the case of the INTERGENOMICS case study). The solutions based on MULTIPLE are more efficient (the development time is reduced) and are more elegant (the level of abstraction is raised). The use of model transfor-

mations eases the automation of manual tasks, such as the generation of *petri nets*, or the definition of the final software architecture in the case of MORPHEUS. Moreover, model transformations are more precise, concise ans easy to understand than traditional and imperative approaches. Serve as an example the INTERGENOMICS case study, where the same transformation is implemented both in QVT and Java (Gómez 2008). In this case, the Java code is ≈ 4.5 times larger than the QVT transformation (407 vs. 1810 lines of code). Besides, declarative model transformations provide traceability capabilities which are fundamental in complex processes.

Finally, the use of standards increases the interoperability and the efficiency, as the most adequate tool can be selected for a precise task. For example, in the case of MORPHEUS the best tool for the pursued purpose is selected. In this case the tool integrates different technologies in a transparent way: .NET, Java, EMF, *MS Office Visio*, etc. The use of standards in this merger of technologies eases the development and allows obtaining a powerful tool with the best of each one.

16 PUBLICATIONS

«Search for the truth is the noblest occupation of man, its publication is a duty.»

— Anne Louise Germaine de Staël-Holstein
French-speaking Swiss writer, 1766–1817

This chapter summarizes the publications which have been produced throughout the realization of this thesis, many of which have already been introduced and referenced in previous chapters.

Prior to the development of this thesis, some fundamental research was done in the context of MDE and formal methods. These works motivate the interest of this thesis in providing model checking capabilities when managing feature models in MMDSPLs:

- A. Boronat, J. Iborra, J. A. Carsí, I. Ramos and A. Gómez (2005c). Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework. In: *Actas X Jornadas sobre Ingeniería del Software y Bases de Datos. JISBD'05. Granada, Spain.* eprint: `http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=26&Itemid=28`

- A. Boronat, J. Iborra, J. A. Carsí, I. Ramos and A. Gómez (2005d). Utilización de Maude desde Eclipse Modeling Frame-

work para la Gestión de Modelos. In: *Desarrollo de Software Dirigido por Modelos - DSDM'05 (Junto a JISBD'05). Granada, Spain*. eprint: `http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=32`

- A. Gómez, A. Boronat, J. A. Carsí and I. Ramos (2005). Integración de un sistema de reescritura de términos en una herramienta de desarrollo software industrial. In: *IV Jornadas de trabajo DYNAMICA. Archena (Murcia)*. eprint: `http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=31`

- A. Gómez, A. Boronat, J. A. Carsí and I. Ramos (2007b). MOMENT-CASE: Un prototipo de herramineta CASE. In: *XII Jornadas de Ingeniería del Software y Bases de Datos. JISBD07. Zaragoza. Spain*. Edited by X. Franch. eprint: `http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=40&Itemid=28`

Query and constraint languages are also an important topic in this thesis. Previous works on constraint languages, and specifically related with OCL were published in the following international conferences:

- A. Boronat, J. Oriente, A. Gómez, I. Ramos and J. A. Carsí (2006b). MOMENT-OCL: Algebraic Specifications of OCL 2.0 within the Eclipse Modeling Framework. In: *ENTCS. Demo. 6th International Workshop on Rewriting Logic and its Applications. Vienna, Austria*. eprint: `http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=42`

- A. Boronat, J. Oriente, A. Gómez, I. Ramos and J. A. Carsí (2006a). An Algebraic Specification of Generic OCL Queries. In: *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006*. Lecture Notes in Computer Science

4066, pages 316–330. eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=22&Itemid=28. URL: http://dx.doi.org/10.1007/11787044_24

DSLs are also a key issue in this thesis. A previous work was done in the development of DSLs and published in the Spanish reference conference on Software Engineering:

- A. Gómez, A. Boronat, L. Hoyos, J. . Carsí and I. Ramos (2006). Definición de operaciones complejas con un lenguaje específico de dominio en Gestión de Modelos. In: *XI Jornadas de Ingeniería del Software y Bases de Datos. JISBD'06. Sitges, Barcelona. Spain*. Edited by J. C. Riquelme and P. Botella. eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=30

The main contribution of this thesis resides in the development of MMDSPLs. In this topic different works have been published in both national and international conferences. Among them we find for example the *Fourth International Workshop on Variability Modelling of Sofware-Intensive Systems*, the most important conference in variability management world-wide. Serve as an example the 2010 edition, which was attended by the most relevant researchers on the area: e. g. K. Kang, K. Czarnecki, D. Batory or D. Benavides among others. It is also noteworthy the publication in the *Information Systems Development* conference evaluated as an A conference in the Computing Research and Education Association of Australasia (CORE) ranking. Works have been also published in the SPL conference and international journals.

- M. E. Cabello, I. Ramos, A. Gómez and R. Limón (2009). Baseline-Oriented Modeling: An MDA Approach Based on Software Product Lines for the Expert Systems Development. In: *Intelligent Information and Database Systems, Asian Conference on* 0, pages 208–213. DOI: http://doi.ieeecomputersociety.org/10.1109/ACIIDS.2009.15

- A. Gómez Llana and I. Ramos Salavert (2010). Cardinality-based feature models and Model-Driven Engineering: fitting them together. In: *Proceedings of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems. ICB-research report No. 37. Pages 61–68. ISSN 1860 - 2770 (Print), ISSN 1866 - 5101 (Online)*. eprint: http://issi.dsic.upv.es/publications/archives/f-1263570538890/document.pdf. URL: http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf

- A. Gómez and I. Ramos (2010). Automatic Tool Support for Cardinality-Based Feature Modeling with Model Constraints for Information Systems Development. In: *19th International Conference on Information Systems Development, Prague, Czech Republic, August 25 - 27, 2010*. eprint: http://issi.dsic.upv.es/publications/archives/f-1285082432517/document.pdf

- A. Gómez, M. E. Cabello and I. Ramos (2010). BOM-Lazy: A Variability-Driven Framework for Software Applications Production Using Model Transformation Techniques. In: *2nd International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010), collocated with the 14th International Software Product Line Conference (SPLC 2010). South Korea, September 2010*. eprint: http://issi.dsic.upv.es/publications/archives/f-1286208702244/document(final).pdf

- M. Gómez Lacruz, A. Gómez Llana, M. E. Cabello Espinosa and I. Ramos Salavert (2009). BOM-Lazy: gestión de la variabilidad en el desarrollo de Sistemas Expertos mediante técnicas de MDA. In: *VI Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM'09). Junto a XIV Jornadas de Ingeniería del Software y Bases de Datos. 8 de Septiembre de 2009, San Sebastián, España.* eprint: http://issi.dsi

c.upv.es/publications/archives/f-1249040813544/dsdm_camera_ready.pdf

- M. E. Cabello, I. Ramos, J. R. Gutiérrez, A. Gómez and r. Limon (2011). SPL variability management, cardinality and types: an MDA approach. In: *International Journal of Intelligent Information and Database Systems*. Unpublished yet, in press

Diverse works have been published with regard to the different case studies provided of the thesis. Works in the bioinformatics field together with MDE have been published both in national and international conferences, besides of international journals such as the IEEE Latin America Transactions:

- A. Gómez, J. A. Carsí, A. Boronat, I. Ramos, C. Täubner and S. Eckstein (2007a). Biological Data Migration Using a Model Driven Approach. In: *4th International Workshop on Software Language Engineering (ateM 2007). Within ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007). Nashville, TN, USA.* Edited by J.-M. Favre, D. Gasevic, R. Lämmel and A. Winter. ISSN: 0931-9972. eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=39&Itemid=28
- A. Gómez, A. Boronat, J. A. Carsí, I. Ramos, C. Täubner and S. Eckstein (2007c). Recuperación y procesado de datos biológicos mediante Ingeniería Dirigida por Modelos. In: *XII Jornadas de Ingeniería del Software y Bases de Datos. JISBD07. Zaragoza. Spain.* Edited by X. Franch. eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=41&Itemid=28. URL: http://www.sistedes.es/sistedes/pdf/2007/JISBD-07-gomez-biologicos.pdf

- A. Gómez, A. Boronat, J. Carsí and I. Ramos (2008). Biological Data Migration in Pathway Simulation. In: *Actas de las VIII Jornadas Nacionales de Bioinformática (JNB'08), Valencia.*

- A. Gomez, A. Boronat, J. A. Carsi, I. Ramos, C. Taubner and S. Eckstein (2008). JISBD2007-03: Biological Data Processing using Model Driven Engineering. In: *Latin America Transactions, IEEE (Revista IEEE America Latina)* 6.4, pages 324–331. ISSN: 1548-0992. DOI: 10.1109/TLA.2008.4815285

With respect to the case study related with software measurement some relevant works have been also published, specially in international conferences (ranked as B in by the CORE) and journals.

- B. Mora, F. Garcia, F. Ruiz, M. Piattini, A. Boronat, A. Gómez, J. Á. Carsí and I. Ramos (2007). Marco de Trabajo basado en MDA para la Medición Genérica del Software. In: *XII Jornadas de Ingeniería del Software y Bases de Datos. JISBD'07. Zaragoza. Spain.* Edited by X. Franch. URL: http://www.sistedes.es/sistedes/pdf/2007/JISBD-07-mora-mda.pdf

- B. Mora, F. García, F. Ruiz, M. Piattini, A. Boronat, A. Gómez, J. A. Carsí and I. Ramos (2008b). Software Measurement by Using QVT Transformations in an MDA Context. In: *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume DISI, Barcelona, Spain, June 12-16, 2008.* Edited by J. Cordeiro and J. Filipe, pages 117–124

- B. Mora, F. Garcia, F. Ruiz, M. Piattini, A. Boronat, A. Gómez, J. Á. Carsí and I. Ramos (2008a). JISBD2007-08: Software generic measurement framework based on MDA. In: *Latin America Transactions, IEEE (Revista IEEE America Latina)* 6.4, pages 363–370. ISSN: 1548-0992. DOI: 10.1109/TLA.2008.4815290

- B. Mora, F. García, F. Ruiz, M. Piattini, A. Boronat, A. Gómez, J. Carsi and I. Ramos (2010). Software Generic Measurement

Framework Based on MDA. In: *Latin America Transactions, IEEE (Revista IEEE America Latina)* 8.5, pages 605 –613. ISSN: 1548-0992. DOI: 10.1109/TLA.2010.5623515

- B. Mora, F. Garcia, F. Ruiz, M. Piattini, A. Boronat, A. Gomez, J. Carsi and I. Ramos (2011). Software Generic Measurement Framework Based on MDA. In: *Latin America Transactions, IEEE (Revista IEEE America Latina)* 9.1, pages 864 –871. ISSN: 1548-0992. DOI: 10.1109/TLA.2011.5876432

Finally, works also in international conferences (CORE A) have been published describing the works on model transformations and software architectures:

- E. Navarro Martínez, A. Gómez Llana, P. Letelier Torres and I. Ramos Salavert (2009). MORPHEUS: a supporting tool for MDD. In: *18th International Conference on Information Systems Development (ISD2009). Nanchang, China. September 16-19, 2009*. eprint: http://issi.dsic.upv.es/publications/archives/f-1249040391367/MORPHEUS_ISD_camera_ready.pdf

APPENDICES

A TRANSFORMATION FEATURES2CLASSDIAGRAM

Listing A.1: Full Features2ClassDiagram transformation

```
1  transformation Feature2ClassDiagram(feature : features,
       classdiagram : \emph{Ecore}) {
2
3    key \emph{Ecore}::EPackage{nsURI};
4    key \emph{Ecore}::EClass{name};
5    key \emph{Ecore}::EReference{name};
6    key \emph{Ecore}::EAnnotation{source, eModelElement};
7    key \emph{Ecore}::EDataType{name};
8
9    top relation Feature2Class {
10
11     checkonly domain feature feature : features::Feature {
12     };
13
14     checkonly domain feature model : features::FeatureModel {
15     };
16
17     enforce domain classdiagram pkg : \emph{Ecore}::EPackage
         {
18       name = model.name,
19       nsPrefix = model.name,
20       nsURI = 'http://' + model.name,
21       eClassifiers = class : \emph{Ecore}::EClass {
22         name = feature.name
23       }
24     };
25   }
26
27   top relation FeatureAttribute2ClassAttribute {
28
29     checkonly domain feature feature : features::Feature {
30       attributes = featureAttribute : features::Attribute {}
31     };
32
33     checkonly domain feature model : features::FeatureModel {
34     };
35
36     enforce domain classdiagram pkg : \emph{Ecore}::EPackage
         {
```

```
37        name = model.name,
38        nsPrefix = model.name,
39        nsURI = 'http://' + model.name,
40        eClassifiers = type : \emph{Ecore}::EDataType {
41          name = featureAttribute.type,
42          instanceTypeName = 'java.lang.' + featureAttribute.
               type
43        }
44      };
45
46      enforce domain classdiagram class : \emph{Ecore}::EClass
           {
47        name = feature.name,
48        eStructuralFeatures = eattribute : \emph{Ecore}::
             EAttribute {
49          name = featureAttribute.name,
50          eType = type
51        }
52      };
53    }
54
55    top relation StructuralRelationship2Reference{
56
57      checkonly domain feature model : features::FeatureModel {
58      };
59
60      checkonly domain feature feature : features::Feature {
61        childs = relationship : features::
             StructuralRelationship {}
62      };
63
64      enforce domain classdiagram pkg : \emph{Ecore}::EPackage
           {
65        nsURI = 'http://' + model.name,
66        eClassifiers = childclass : \emph{Ecore}::EClass {
67          name = relationship.to.name
68        }
69      };
70
71      enforce domain classdiagram class : \emph{Ecore}::EClass
           {
72        name = feature.name,
73        eStructuralFeatures = reference : \emph{Ecore}::
             EReference {
74          name = relationship.to.name,
75          eType = childclass,
76          upperBound = relationship.upperBound,
77          lowerBound = relationship.lowerBound,
78          containment = true
79        }
```

```
80      };
81
82    }
83
84    top relation Group2Reference {
85
86
87      checkonly domain feature model : features::FeatureModel {
88      };
89
90      checkonly domain feature feature : features::Feature {
91        group = group : features::Group {
92          childs = childRelationship : features::
                 StructuralRelationship {
93            to = childFeature : features::Feature {}
94          }
95        }
96      };
97
98      enforce domain classdiagram pkg : \emph{Ecore}::EPackage
            {
99        nsURI = 'http://' + model.name,
100       eClassifiers = typeClass : \emph{Ecore}::EClass {
101         name = group.name,
102         abstract = true
103       },
104       eClassifiers = parentClass : \emph{Ecore}::EClass {
105         name = feature.name,
106         eStructuralFeatures = reference : \emph{Ecore}::
                 EReference {
107           name = feature.name + 'Features',
108           eType = typeClass,
109           upperBound = -1,
110           lowerBound = 0,
111           containment = true
112         }
113       },
114       eClassifiers = childClass : \emph{Ecore}::EClass {
115         name = childFeature.name
116       }
117     };
118
119     where {
120       GroupChild2Classes(childFeature, typeClass, childClass)
             ;
121       GroupChild2ChildrenAnnot(feature, parentClass);
122       GroupChild2LowerAnnot(childRelationship, parentClass);
123       GroupChild2UpperAnnot(childRelationship, parentClass);
124     }
125   }
```

```
126
127   relation GroupChild2Classes {
128
129     checkonly domain feature feature : features::Feature {};
130
131     checkonly domain classdiagram typeClass : \emph{Ecore}::
            EClass {
132     };
133     enforce domain classdiagram parentclass : \emph{Ecore}::
            EClass {
134       name = feature.name,
135       eSuperTypes = eSuperTypes->including(typeClass)
136     };
137
138   }
139
140   relation GroupChild2ChildrenAnnot {
141
142     checkonly domain feature feature : features::Feature {
143     };
144
145     enforce domain classdiagram parentClass : \emph{Ecore}::
            EClass {
146         eAnnotations = oclAnnotLower : \emph{Ecore}::
                EAnnotation {
147           source = 'http://www.eclipse.org/ocl/examples/OCL',
148           details = oclEntryLower : \emph{Ecore}::
                  EStringToStringMapEntry {
149             _key = 'checkChildren' + feature.name,
150             value = toString(feature.group.lowerBound) + ' <=
                     ( ' + buildGroupConstraint(feature) + ' )
                    and ( ' +
151                 buildGroupConstraint(feature) + ' ) <= ' +
                        toString(feature.group.upperBound)
152           }
153         }
154     };
155     when {
156       parentClass.eAnnotations->select(
157           annot : \emph{Ecore}::EAnnotation | not annot.
                  details->select(
158             entry : \emph{Ecore}::EStringToStringMapEntry |
                    entry._key = 'checkChildren' + feature.name)
                    ->isEmpty())->isEmpty();
159     }
160   }
161
162   relation GroupChild2LowerAnnot {
163
```

```
164     checkonly domain feature relationship : features::
            StructuralRelationship {
165       from = group : features::Group {},
166       to = feature : features::Feature {}
167     };
168
169     enforce domain classdiagram parentClass : \emph{Ecore}::
            EClass {
170       eAnnotations = oclAnnotLower : \emph{Ecore}::
              EAnnotation {
171         source = 'http://www.eclipse.org/ocl/examples/OCL',
172         details = oclEntryLower : \emph{Ecore}::
                EStringToStringMapEntry {
173           _key = 'lowerMultiplicity' + feature.name,
174           value = 'self.' + group.parentFeature.name + '
                  Features->select(f : ' + group.name + ' | f.
                  oclIsKindOf(' + feature.name + '))->notEmpty()
                  implies ' +
175           'self.' + group.parentFeature.name + 'Features->
                  select(f : ' + group.name + ' | f.oclIsKindOf('
                   + feature.name + '))->size() >=' + toString(
                  relationship.lowerBound)
176         }
177       }
178     };
179     when {
180       parentClass.eAnnotations->select(
181           annot : \emph{Ecore}::EAnnotation | not annot.
                  details->select(
182             entry : \emph{Ecore}::EStringToStringMapEntry |
                    entry._key = 'lowerMultiplicity' + feature.
                    name)->isEmpty())->isEmpty();
183     }
184   }
185
186   relation GroupChild2UpperAnnot {
187
188     checkonly domain feature relationship : features::
            StructuralRelationship {
189       from = group : features::Group {},
190       to = feature : features::Feature {}
191     };
192
193     enforce domain classdiagram parentClass : \emph{Ecore}::
            EClass {
194       eAnnotations = oclAnnotUpper : \emph{Ecore}::
              EAnnotation {
195         source = 'http://www.eclipse.org/ocl/examples/OCL',
196         details = oclEntryUpper : \emph{Ecore}::
                EStringToStringMapEntry {
```

```
197            _key = 'upperMultiplicity' + feature.name,
198            value = 'self.' + group.parentFeature.name + '
                   Features->select(f : ' + group.name + ' | f.
                   oclIsKindOf(' + feature.name + '))->notEmpty()
                   implies ' +
199            'self.' + group.parentFeature.name + 'Features->
                   select(f : ' + group.name + ' | f.oclIsKindOf('
                    + feature.name + '))->size() <=' + toString(
                   relationship.upperBound)
200          }
201        }
202      };
203      when {
204        -- The rule is only executed when upperBound is > 0, i.
               e.,
205        -- we only must create the restriction when there is an
                upper bound
206        relationship.upperBound > 0
207        and
208        parentClass.eAnnotations->select(
209            annot : \emph{Ecore}::EAnnotation | not annot.
                   details->select(
210              entry : \emph{Ecore}::EStringToStringMapEntry |
                     entry._key = 'upperMultiplicity' + feature.
                     name)->isEmpty())->isEmpty();
211      }
212    }
213
214    top relation UsesRelationship2Reference{
215
216      checkonly domain feature model : features::FeatureModel {
217        relationships = usesRelationship : features::Uses {
218          from = fromFeature : features::Feature {},
219          to = toFeature : features::Feature {}
220        }
221      };
222
223      enforce domain classdiagram pkg : \emph{Ecore}::EPackage
             {
224        nsURI = 'http://' + model.name,
225        eClassifiers = toClass : \emph{Ecore}::EClass {
226          name = toFeature.name
227        }
228      };
229
230      enforce domain classdiagram pkg : \emph{Ecore}::EPackage
             {
231        nsURI = 'http://' + model.name,
232        eClassifiers = fromClass : \emph{Ecore}::EClass {
233          name = fromFeature.name,
```

```
234         eStructuralFeatures = reference : \emph{Ecore}::
                EReference {
235           name = usesRelationship.name,
236           eType = toClass,
237           upperBound = usesRelationship.upperBound,
238           lowerBound = usesRelationship.lowerBound,
239           containment = false
240         }
241       }
242     };
243 
244     where {
245       if (not usesRelationship.opposite.oclIsUndefined())
              then
246         UsesRelationship2EOppositeReference(usesRelationship.
                opposite, reference)
247       else
248         true
249       endif;
250     }
251   }
252 
253   relation UsesRelationship2EOppositeReference {
254 
255     checkonly domain feature opposite : features::Uses {
256     };
257 
258     enforce domain classdiagram reference : \emph{Ecore}::
            EReference {
259       eOpposite = oppositeReference : \emph{Ecore}::
              EReference {
260         name = opposite.name
261       }
262     };
263   }
264 
265   top relation ExcludesRelationship2ModelConstraint{
266 
267     checkonly domain feature model : features::FeatureModel {
268       relationships = excludesRelationship : features::
              Excludes {
269         from = fromFeature : features::Feature {},
270         to = toFeature : features::Feature {}
271       }
272     };
273 
274     enforce domain classdiagram pkg : \emph{Ecore}::EPackage
            {
275       nsURI = 'http://' + model.name,
```

```
276        eAnnotations = oclAnnotExcludes: \emph{Ecore}::
               EAnnotation {
277          source = 'http://www.eclipse.org/ocl/examples/OCL',
278          details = detailExcludes: \emph{Ecore}::
                 EStringToStringMapEntry {
279            _key = fromFeature.name + '_exclusion_' + toFeature
                   .name,
280            value = '(' + fromFeature.name + '.allInstances()->
                   notEmpty() implies ' + toFeature.name + '.
                   allInstances()->isEmpty()) and ' +
281                   '(' + toFeature.name + '.allInstances()->
                       notEmpty() implies ' + fromFeature.name + '
                       .allInstances()->isEmpty())'
282          }
283        }
284      };
285    }
286
287    top relation BiconditionalRelationship2ModelConstraint{
288
289      checkonly domain feature model : features::FeatureModel {
290        relationships = biconditionalRelationship : features::
               Biconditional {
291          from = fromFeature : features::Feature {},
292          to = toFeature : features::Feature {}
293        }
294      };
295
296      enforce domain classdiagram pkg : \emph{Ecore}::EPackage
             {
297        nsURI = 'http://' + model.name,
298        eClassifiers = fromClass : \emph{Ecore}::EClass {
299          name = fromFeature.name,
300          eAnnotations = oclAnnotFrom: \emph{Ecore}::
                 EAnnotation {
301            source = 'http://www.eclipse.org/ocl/examples/OCL',
302            details = detailFrom : \emph{Ecore}::
                   EStringToStringMapEntry {
303              _key = fromFeature.name + '_biconditional_' +
                     toFeature.name,
304              value = toFeature.name + '.allInstances()->
                     notEmpty()'
305            }
306          }
307        },
308        eClassifiers = toClass : \emph{Ecore}::EClass {
309          name = toFeature.name,
310          eAnnotations = oclAnnotTo : \emph{Ecore}::EAnnotation
                 {
311            source = 'http://www.eclipse.org/ocl/examples/OCL',
```

```
312          details = detailTo : \emph{Ecore}::
                 EStringToStringMapEntry {
313            _key = toFeature.name + '_biconditional_' +
                 fromFeature.name,
314            value = fromFeature.name + '.allInstances()->
                 notEmpty()'
315          }
316        }
317      }
318    };
319  }
320
321  top relation ImpliesRelationship2ModelConstraint{
322
323    checkonly domain feature model : features::FeatureModel {
324      relationships = impliesRelationship : features::Implies
           {
325        from = fromFeature : features::Feature {},
326        to = toFeature : features::Feature {}
327      }
328    };
329
330    enforce domain classdiagram pkg : \emph{Ecore}::EPackage
         {
331      nsURI = 'http://' + model.name,
332      eClassifiers = fromClass : \emph{Ecore}::EClass {
333        name = fromFeature.name,
334        eAnnotations = oclAnnotFrom: \emph{Ecore}::
             EAnnotation {
335          source = 'http://www.eclipse.org/ocl/examples/OCL',
336          details = detailFrom : \emph{Ecore}::
                 EStringToStringMapEntry {
337            _key = fromFeature.name + '_implies_' + toFeature
                 .name,
338            value = toFeature.name + '.allInstances()->
                 notEmpty()'
339          }
340        }
341      }
342    };
343  }
344
345  top relation FMCLConstraint2OCLConstraint{
346
347    checkonly domain feature model : features::FeatureModel {
348      modelConstraints = modelConstraints : features::
           ConstraintsSet {
349        _context = _context : features::ConstrainableElement
             {},
350        constraints = constraint : features::Constraint {}
```

```
351       }
352     };
353 
354     enforce domain classdiagram pkg : \emph{Ecore}::EPackage
          {
355       nsURI = 'http://' + model.name,
356       eClassifiers = fromClass : \emph{Ecore}::EClass {
357         name = _context.oclAsType(Feature).name,
358         eAnnotations = oclAnnotFrom: \emph{Ecore}::
              EAnnotation {
359           source = 'http://www.eclipse.org/ocl/examples/OCL',
360           details = detailFrom : \emph{Ecore}::
                EStringToStringMapEntry {
361             _key = constraint.name,
362             value = translateFMCLtoOCL(constraint._body)
363           }
364         }
365       }
366     };
367   }
368 
369 
370 
371   query toString(number : Integer) : String {
372     -- We define the following expression to translate an
          Integer to String.
373     -- In this way, we avoid to include any external library/
          method to perform
374     -- the conversion.
375     if number >= 0 then
376       OrderedSet{1000000, 10000, 1000, 100, 10, 1}->iterate(
377       -- We will supports numbers <= 999.999
378       -- If greater numbers are needed, more powers of ten
            can be added
379         denominator : Integer;
380         s : String = ''|
381         let numberAsString : String = OrderedSet{'0','1','2',
              '3','4','5','6','7','8','9'}
382         ->at(((number div denominator) mod 10) + 1)
383         in
384           if s='' and numberAsString = '0' then
385             s
386           else
387             s.concat(numberAsString)
388           endif
389           )
390     else
391       '-'.concat(toString(-number))
392     endif
393   }
```

```
394
395     query buildGroupConstraint(parentFeature : Feature) :
            String {
396       parentFeature.group.childs->iterate(
397         -- We iterate for each relationship contained in the
              group
398         relationship : StructuralRelationship;
399         -- The text of the OCL expression is stored in the "
              result" var
400         result : String = ''
401         | -- Starting from here, the body of the loop
402         result.concat(
403             '(if self.' + parentFeature.name + 'Features->
                  select(f : ' + parentFeature.group.name + ' | f
                  .oclIsKindOf(' + relationship.to.name + '))->
                  notEmpty() then 1 else 0 endif) + ')
404         ).concat('0')
405     }
406
407     query translateFMCLtoOCL(expression : String) : String {
408
409
410       ConstrainableElement.allInstances()->iterate(
411         elt : ConstrainableElement;
412         s : String = expression |
413         -- The order when applying the substitutions is
              important
414         -- We must go from the most specific case to the most
              general one
415         if (elt.oclIsTypeOf(features::Feature)) then
416           s.replace('(' + elt.name + '\b)\.childs\(\)', '$1Type
                .allInstances()')
417         else
418           s
419         endif
420         .replace('(' + elt.name + '\b)\.(\w+\s+\S+\s+.+)', '$1.
              allInstances()->forAll($2)')
421         .replace('(' + elt.name + '\b)\.selected\(\)', '$1.
              allInstances()->notEmpty()')
422         .replace('(' + elt.name + '\b)(?:\.allInstances\(\))?',
               '$1.allInstances()')
423         )
424     }
425
426   }
```

B TRANSFORMATION MODULES2COMPONENTS

Listing B.1: Full Modules2Components transformation

```
1  transformation modules2components(mdomain : mview,  dcmdomain
       : dcm, ccdomain : ccview) {
2
3    key ccview::Component{name};
4    key ccview::Connector{name};
5    key ccview::Port{name,powner};
6    key ccview::Role{name,cowner};
7    key ccview::Service{name,sowner};
8    key ccview::PeerToPeer{name,service};
9
10   top relation ModulesModel2ComponentsModel {
11
12     checkonly domain mdomain modulesModel : mview::
           ModulesModel {};
13
14     checkonly domain dcmdomain varModel : dcm::
           DomainConceptualModel {};
15
16     enforce domain ccdomain componentsModel : ccview::CCModel
            {
17       name = modulesModel.name
18     };
19
20     where {
21       UseCase2Connector(modulesModel,varModel,componentsModel
             );
22     }
23   }
24
25
26   relation UseCase2Connector {
27
28     checkonly domain mdomain modulesModel : mview::
           ModulesModel {};
29
30     checkonly domain dcmdomain varModel : dcm::
           DomainConceptualModel {
31       useCase = useCase : dcm::UseCase {}
32     };
```

```
33
34      enforce domain ccdomain componentsModel : ccview::CCModel
             {
35        tcomponents = connector : ccview::Connector {
36          name = useCase.name + ' Connector'
37        }
38      };
39
40      where {
41        Module2Component(modulesModel, varModel,
               componentsModel, connector, useCase);
42      }
43    }
44
45
46    relation Module2Component {
47
48      checkonly domain mdomain modulesModel : mview::
            ModulesModel {
49        tmodules = module : mview::Module {}
50      };
51
52      checkonly domain dcmdomain varModel : dcm::
            DomainConceptualModel {
53        Actor = actor : dcm::Actor {
54          uses_UseCase = useCaseActor : dcm::UseCase {}
55        }
56      };
57
58      enforce domain ccdomain componentsModel : ccview::CCModel
             {
59        tcomponents = component : ccview::Component {
60          name = getComponentName(varModel, actor, module)
61        }
62      };
63
64      enforce domain ccdomain connector : ccview::Connector {};
65
66      checkonly domain dcmdomain useCase : dcm::UseCase {};
67
68
69      where {
70        if module.name = 'User Interface' then
71          Module2RolePort(module, useCaseActor, connector,
                 component)
72        else
73          Module2RolePort(module, useCase, connector, component
                 )
74        endif;
75
```

```
76      Function2Relation(module, componentsModel, connector,
            component);
77    }
78  }
79
80  relation Module2RolePort {
81
82    checkonly domain mdomain module : mview::Module {};
83
84    checkonly domain dcmdomain useCase : dcm::UseCase {};
85
86    enforce domain ccdomain connector : ccview::Connector {
87      crole = role : ccview::Role {
88        name = module.name + ' Role'
89      }
90    };
91
92    enforce domain ccdomain component : ccview::Component {
93      port = port : ccview::Port {
94        name = useCase.name + ' Port'
95      }
96    };
97
98    where {
99      ConnectRoleAndPort(role, port);
100     if (connector.name.oclIsUndefined() or connector.name =
            useCase.name + ' Connector') then
101       Function2Service(module, port, component)
102     else
103       true
104     endif;
105   }
106 }
107
108 relation ConnectRoleAndPort {
109
110   checkonly domain ccdomain role : ccview::Role {};
111
112   enforce domain ccdomain port : ccview::Port {
113     prole = role
114   };
115
116   when {
117     role.rport.oclIsUndefined() and port.prole.
            oclIsUndefined();
118   }
119 }
120
121 relation Function2Service {
122
```

```
123     checkonly domain mdomain module : mview::Module {
124       function = function : mview::Function {}
125     };
126
127     checkonly domain ccdomain port : ccview::Port {};
128
129     enforce domain ccdomain component: ccview::Component {
130       cservice = service : ccview::Service {
131         name = function.name,
132         type = function.type,
133         port = port
134       }
135     };
136   }
137
138
139   relation Function2Relation {
140
141     checkonly domain mdomain module : mview::Module {
142       function = function : mview::Function {}
143     };
144
145     enforce domain ccdomain componentsModel : ccview::CCModel
          {
146       relations = relat : ccview::PeerToPeer {
147         name = module.name + ' Attachment',
148         type = 'attachment',
149         service = function.name,
150         constraints = function.type,
151         component = component,
152         connector = connector
153       }
154     };
155
156     checkonly domain ccdomain connector : ccview::Connector
          {};
157
158     checkonly domain ccdomain component: ccview::Component
          {};
159
160   }
161
162   query getComponentName(varModel : dcm::
        DomainConceptualModel, actor : dcm::Actor, module :
        mview::Module) : String {
163
164     if module.name = 'User Interface' then
165       module.name + ' - ' + actor.name
166     else
```

```
167         if module.name = 'Inference Motor' or module.name = '
              Knowledge Base'  then
168           if varModel.Reasoning.ReasoningFeatures.oclIsTypeOf(
                dcm::deductive) then
169             'Deductive ' + module.name
170           else
171             if varModel.Reasoning.ReasoningFeatures.oclIsTypeOf
                  (dcm::differential) then
172               'Differential ' + module.name
173             else
174               'Error ' + module.name
175             endif
176           endif
177         else
178           module.name
179         endif
180       endif
181     }
182   }
```

C TRANSFORMATION COMPONENTS2PRISMA

Listing C.1: Full Components2Prisma transformation

```
1  transformation Components2Prisma(adcmdomain : adcm, cdomain :
       ccview, prismadomain: prisma) {
2
3
4    key prisma::PRISMAArchitecture{name};
5    key prisma::Component{name};
6    key prisma::Connector{name};
7    key prisma::Aspect{name};
8
9
10   top relation CCModel2PRISMAArchitecture {
11
12     checkonly domain adcmdomain adcm: adcm::
           AppDomainConceptualModel {};
13
14     checkonly domain cdomain model : ccview::CCModel {};
15
16     enforce domain prismadomain arch : prisma::
           PRISMAArchitecture {
17       name = model.name
18     };
19
20     where {
21       Component2Component(adcm, model, arch);
22     }
23   }
24
25   relation Component2Component {
26
27     checkonly domain adcmdomain adcm: adcm::
           AppDomainConceptualModel {};
28
29     checkonly domain cdomain model : ccview::CCModel {
30       tcomponents = ccomponent : ccview::Component {}
31     };
32
33     enforce domain prismadomain arch: prisma::
           PRISMAArchitecture {
34       includes = pcomponent : prisma::System {
```

```
35          name = ccomponent.name
36        }
37      };
38
39      where {
40        Port2PortInterface(arch, adcm, ccomponent, pcomponent);
41      }
42    }
43
44
45    relation Port2PortInterface {
46
47      checkonly domain prismadomain arch: prisma::
           PRISMAArchitecture {};
48
49      checkonly domain adcmdomain adcm: adcm::
           AppDomainConceptualModel {};
50
51      checkonly domain cdomain ccomponent : ccview::Component {
52        port = cport : ccview::Port {
53          prole = prole : ccview::Role {}
54        }
55      };
56
57      enforce domain prismadomain pcomponent : prisma::System {
58        has = pport : prisma::Port {
59          name = cport.name,
60          typed = interface :prisma::Interface {
61            name = 'I' + ccomponent.name
62          }
63        }
64      };
65
66      where {
67        Component2FunctionalAspect(adcm, ccomponent, pport,
             interface, pcomponent);
68        Connector2ConnectorPortInterface(pport, pcomponent,
             prole, adcm, ccomponent, arch);
69      }
70    }
71
72    relation Component2FunctionalAspect {
73
74      checkonly domain adcmdomain adcm : adcm::
           AppDomainConceptualModel {};
75
76      checkonly domain cdomain ccomponent : ccview::Component {
77        cservice = cservice : ccview::Service {}
78      };
79
```

```
80      checkonly domain prismadomain pport : prisma::Port {};
81
82      checkonly domain prismadomain interface : prisma::
            Interface {};
83
84      enforce domain prismadomain pcomponent : prisma::
            Component {
85        imports = aspect : prisma::Aspect {
86          name = 'F' + ccomponent.name,
87          using = using->including(interface),
88          providesRequires = service :prisma::Service {
89            name = cservice.name,
90            servType = cservice.type,
91            isInInterface = isInInterface->including(interface)
92          }
93        }
94      };
95
96      where {
97        Propery2Parameter(adcm, service);
98        Hypotheses2Parameter(adcm, service);
99        AddServices2Interface(service, interface);
100       AddPlayedRole2Aspect(pport, interface, aspect);
101       Property2ConstantAttribute(adcm, aspect);
102       Hypotheses2VariableAttribute(adcm, aspect);
103       Rule2DerivedAttribute(adcm, aspect);
104     }
105   }
106
107   relation Propery2Parameter {
108
109     checkonly domain adcmdomain adcm: adcm::
            AppDomainConceptualModel {
110       properties = property : adcm::Property {}
111     };
112
113     enforce domain prismadomain service : prisma::Service {
114       has = parameter : prisma::Parameter {
115         name = property.name,
116         paramKind = property.type
117       }
118     };
119   }
120
121   relation Hypotheses2Parameter {
122
123     checkonly domain adcmdomain adcm: adcm::
            AppDomainConceptualModel {
124       hypotheses = hyp : adcm::Hypotheses {}
125     };
```

```
126
127     enforce domain prismadomain service : prisma::Service {
128       has = parameter : prisma::Parameter {
129         name = hyp.name,
130         paramKind = hyp.type
131       }
132     };
133   }
134
135   relation AddServices2Interface {
136
137     checkonly domain prismadomain service : prisma::Service
          {};
138
139     enforce domain prismadomain interface : prisma::Interface
           {
140           publish = publish->including(service)
141     };
142   }
143
144
145   relation AddPlayedRole2Aspect {
146
147     checkonly domain prismadomain pport : prisma::Port {};
148
149     checkonly domain prismadomain interface : prisma::
          Interface {};
150
151     enforce domain prismadomain aspect: prisma::Aspect {
152       plays = plays : prisma::PlayedRole {
153         name = 'Played_role_' + pport.name,
154         for = interface
155       }
156     };
157
158     where {
159       AddPlayedRole2Port(plays, pport);
160     }
161   }
162
163   relation AddPlayedRole2Port {
164
165     checkonly domain prismadomain plays : prisma::PlayedRole
          {};
166
167     enforce domain prismadomain pport : prisma::Port {
168       behaves = plays
169     };
170
171   }
```

```
172
173
174  relation Property2ConstantAttribute {
175
176    checkonly domain adcmdomain adcm: adcm::
         AppDomainConceptualModel {
177      properties = property : adcm::Property {}
178    };
179
180    enforce domain prismadomain aspect : prisma::Aspect {
181      described_by = attribute : prisma::ConstantAttribute {
182        name = property.name,
183        value = property.value,
184        type = property.type
185      }
186    };
187  }
188
189
190  relation Hypotheses2VariableAttribute {
191
192    checkonly domain adcmdomain adcm: adcm::
         AppDomainConceptualModel {
193      hypotheses = hyp : adcm::Hypotheses {}
194    };
195
196    enforce domain prismadomain aspect : prisma::Aspect {
197      described_by = attribute : prisma::VariableAttribute {
198        name = hyp.name,
199        value = hyp.value,
200        type = hyp.value
201      }
202    };
203  }
204
205  relation Rule2DerivedAttribute {
206
207    checkonly domain adcmdomain adcm: adcm::
         AppDomainConceptualModel {
208      rules = rule : adcm::Rule {}
209    };
210
211    enforce domain prismadomain aspect : prisma::Aspect {
212      described_by = attribute : prisma::DerivedAttribute {
213        name = rule.name,
214        complexTerm = rule.clause
215      }
216    };
217
218  }
```

```
219
220  relation Connector2ConnectorPortInterface {
221
222    checkonly domain prismadomain pport : prisma::Port {};
223
224    checkonly domain prismadomain pcomponent : prisma::System
           {};
225
226    checkonly domain cdomain prole : ccview::Role {
227      cowner = cconnector : ccview::Connector {}
228    };
229
230    checkonly domain adcmdomain adcm: adcm::
         AppDomainConceptualModel {};
231
232    checkonly domain cdomain ccomponent : ccview::Component
         {};
233
234    enforce domain prismadomain arch: prisma::
         PRISMAArchitecture {
235      synchronizes = pconnector : prisma::Connector {
236        name = cconnector.name,
237        has = role : prisma::Port {
238          name = prole.name,
239          typed = interface :prisma::Interface {
240            name = 'I' + pconnector.name + '-' + role.name
241          }
242        }
243      }
244    };
245    where {
246      ConnectRolePort(arch, pport, ccomponent, role,
           pcomponent);
247      Connector2CoordinatorAspect(adcm, role, interface,
           cconnector, pconnector);
248    }
249  }
250
251  relation ConnectRolePort {
252
253    checkonly domain prismadomain arch: prisma::
         PRISMAArchitecture {};
254
255    checkonly domain prismadomain pport : prisma::Port {};
256
257    checkonly domain cdomain ccomponent : ccview::Component {
258      clink = clink : ccview::Relation {}
259    };
260
261    checkonly domain prismadomain role : prisma::Port {};
```

```
262
263     enforce domain prismadomain pcomponent : prisma::System {
264       hasLinks = link : prisma::LinkElement {
265         name = clink.name,
266         sourcePort = pport,
267         targetPort = role
268       }
269     };
270
271     where {
272       AddAttachmentsBindingsToPortAndArc(link, pport, arch);
273     }
274   }
275
276   relation AddAttachmentsBindingsToPortAndArc {
277
278     checkonly domain prismadomain link : prisma::LinkElement
          {};
279
280     enforce domain prismadomain pport : prisma::Port {
281       attachmentsBindings = attachmentsBindings->including(
            link)
282     };
283
284     enforce domain prismadomain arch: prisma::
          PRISMAArchitecture {
285       connects = attachment : prisma::Attachment {
286         name = link.name,
287         sourcePort = link.sourcePort,
288         targetPort = link.targetPort
289       }
290     };
291
292   }
293
294   relation Connector2CoordinatorAspect {
295
296     checkonly domain adcmdomain adcm : adcm::
          AppDomainConceptualModel {};
297
298     checkonly domain prismadomain role : prisma::Port {};
299
300     checkonly domain prismadomain interface : prisma::
          Interface {};
301
302     checkonly domain cdomain cconnector : ccview::Connector
          {};
303
304     enforce domain prismadomain pconnector : prisma::
          Connector {
```

```
305        imports = aspect : prisma::Aspect {
306          name = 'C' + cconnector.name,
307          using = using->including(interface)
308        }
309      };
310
311      where {
312        AddPlayedRole2Aspect(role, interface, aspect);
313        Property2ConstantAttribute(adcm, aspect);
314        Hypotheses2VariableAttribute(adcm, aspect);
315        Rule2DerivedAttribute(adcm, aspect);
316      }
317
318    }
319
320  }
```

TRANSFORMATION MULTIPLEFEATURES2FAMAFEATURES

Listing D.1: MULTIPLE Features to FAMA features transformation

```
1  transformation MultipleFeatures2FamaFeatures(mdomain :
       features, fdomain : FeatureModelSchema){
2
3    key BinaryRelationType {name};
4    key SetRelationType {name,cardinality};
5    key GeneralFeature {name};
6    key FeatureModelType {feature};
7
8    top relation Model2Model {
9
10     checkonly domain mdomain fmodel : features::FeatureModel
           {
11       rootFeature = root : features::Feature {}
12     };
13     enforce domain fdomain model : FeatureModelSchema::
           FeatureModelType {
14       feature = first : FeatureModelSchema::GeneralFeature {
15         name = root.name
16       }
17     };
18   }
19
20   top relation StructuralRelationship2BinaryRelation {
21
22     checkonly domain mdomain model : features::FeatureModel {
23     };
24
25     checkonly domain mdomain mfeature : features::Feature {
26       childs = relationship : features::
             StructuralRelationship {}
27     };
28
29     enforce domain fdomain ffeature : FeatureModelSchema::
           GeneralFeature {
30       name = mfeature.name,
31       binaryRelation = binaryRelation : FeatureModelSchema::
             BinaryRelationType {
32         name = 'Relation_to_' + relationship.to.name,
```

```
33          cardinality = cardinality : FeatureModelSchema::
                CardinalityType {
34            max = relationship.upperBound,
35            min = relationship.lowerBound
36          },
37          solitaryFeature = generalfeature : FeatureModelSchema
                ::GeneralFeature {
38            name = relationship.to.name
39          }
40        }
41      };
42    }
43
44    top relation Group2SetRelation {
45      checkonly domain mdomain model : features::FeatureModel {
46      };
47
48      checkonly domain mdomain feature : features::Feature {
49        group = group : features::Group {
50          childs = childRelationship : features::
                StructuralRelationship {
51          }
52        }
53      };
54      enforce domain fdomain feature2 : FeatureModelSchema::
            GeneralFeature {
55        name = feature.name,
56        setRelation = setRelation : FeatureModelSchema::
              SetRelationType {}
57      };
58      enforce domain fdomain setRelation : FeatureModelSchema::
            SetRelationType {
59          name = 'Grouped_Relation',
60          cardinality= cardinality : FeatureModelSchema::
                CardinalityType{
61          },
62          groupedFeature = generalfeatures : FeatureModelSchema
                ::GeneralFeature {
63            name = childRelationship.to.name
64          }
65      };
66      enforce domain fdomain cardinality : FeatureModelSchema::
            CardinalityType{
67            max = group.upperBound,
68            min = group.lowerBound
69      };
70    }
71
72    top relation ExcludesRelationship2ExcludesType{
73
```

```
74     checkonly domain mdomain model : features::FeatureModel {
75       relationships = excludesRelationship : features::
             Excludes {
76         from = fromFeature : features::Feature {},
77         to = toFeature : features::Feature {}
78       }
79     };
80
81     enforce domain fdomain featureModel : FeatureModelSchema
           ::FeatureModelType {
82       feature = feature : FeatureModelSchema::GeneralFeature
             {
83         name = model.rootFeature.name
84       },
85       excludes= exclude : FeatureModelSchema::ExcludesType {
86         name= 'Excludes_from_' + from.name + '_to_' + to.name
               ,
87         excludes=from.name,
88         feature=to.name
89       }
90
91     };
92   }
93
94   top relation ImpliesRelationship2RequiresType{
95     checkonly domain mdomain model : features::FeatureModel {
96       relationships = requiresRelationship :features::Implies
              {
97         from = fromFeature : features::Feature {},
98         to = toFeature : features::Feature {}
99       }
100    };
101    enforce domain fdomain featureModel : FeatureModelSchema
           ::FeatureModelType {
102      feature = feature :FeatureModelSchema::GeneralFeature {
103        name = model.rootFeature.name
104      },
105      requires= require:FeatureModelSchema::RequiresType {
106        name = from.name + '_requires_' + to.name,
107        requires = from.name,
108        feature = to.name
109      }
110
111    };
112
113  }
114 }
```

E

FAMA XML SCHEMA DEFINITION

Listing E.1: FAMA XML Schema Definition

```
1
2
3 <?xml version="1.0" encoding="UTF-8"?>
4 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
      elementFormDefault="qualified">
5   <!-- a feature model coprises a root and 0..* constraints
        -->
6   <xs:element name="feature-model">
7     <xs:complexType>
8       <xs:sequence>
9         <xs:element ref="feature"/>
10        <xs:element ref="requires" minOccurs="0" maxOccurs="
              unbounded"/>
11        <xs:element ref="excludes" minOccurs="0" maxOccurs="
              unbounded"/>
12      </xs:sequence>
13    </xs:complexType>
14  </xs:element>
15  <!-- a root is comprises with 0..* relations that can be
        either binary or set relations-->
16  <!-- a general relation has an atribute name to indentify
        it -->
17  <xs:complexType name="generalRelation">
18    <xs:attribute name="name" use="required"/>
19  </xs:complexType>
20  <!-- a relation is of the type generalRelation -->
21  <xs:element name="relation" type="generalRelation"/>
22  <!-- a binary relation is of an extended type of
        generalRelation and comprises only one solitaryFeature
        -->
23  <xs:element name="binaryRelation">
24    <xs:complexType>
25      <xs:complexContent>
26        <xs:extension base="generalRelation">
27          <xs:sequence>
28            <xs:element ref="cardinality" />
29            <xs:element ref="solitaryFeature" />
30          </xs:sequence>
31        </xs:extension>
```

```
32        </xs:complexContent>
33      </xs:complexType>
34    </xs:element>
35    <!-- a set relation is of an extended type of
          generalRelation and comprises 0..* cadninalities and
          2..* groupedFeature -->
36    <xs:element name="setRelation">
37      <xs:complexType>
38        <xs:complexContent>
39          <xs:extension base="generalRelation">
40            <xs:sequence>
41              <xs:element ref="cardinality" maxOccurs="
                    unbounded"/>
42              <xs:element ref="groupedFeature" minOccurs="2"
                    maxOccurs="unbounded"/>
43            </xs:sequence>
44          </xs:extension>
45        </xs:complexContent>
46      </xs:complexType>
47    </xs:element>
48    <!-- a generalFeature is a type that has an attribute
          called name to identified it and comprises 0..*
          relations, it also has an element called attribute -->
49    <xs:complexType name="generalFeature">
50      <xs:sequence minOccurs="0" maxOccurs="unbounded">
51          <xs:element ref="binaryRelation" minOccurs="0"
                maxOccurs="unbounded"/>
52          <xs:element ref="setRelation" minOccurs="0" maxOccurs
                ="unbounded"/>
53      </xs:sequence>
54      <xs:attribute name="name" use="required"/>
55    </xs:complexType>
56
57    <!-- a feature is of the type generalFeature -->
58    <xs:element name="feature" type="generalFeature"/>
59
60    <!--  solitaryFeature is of an extended type of
          generalFeature and comprises of 1..* cardinalities -->
61    <xs:element name="solitaryFeature" type="generalFeature"/>
62
63    <!--  groupedFeature is of an extended type of
          generalFeature  and has implicitly the cardinality
          [1..1]-->
64    <xs:element name="groupedFeature" type="generalFeature"/>
65    <!-- a cardinality comprises two attributes called min and
          max indicating the boundaries of the cardinalities-->
66    <xs:element name="cardinality">
67      <xs:complexType>
68        <xs:attribute name="min" use="required"/>
69        <xs:attribute name="max" use="required"/>
```

```
70     </xs:complexType>
71   </xs:element>
72   <!-- requires and  excludes constraints  an attribute
         called name to identify it-->
73   <xs:element name="requires">
74     <xs:complexType>
75       <xs:attribute name="name" use="required"/>
76       <xs:attribute name="feature" use="required"/>
77       <xs:attribute name="requires" use="required"/>
78     </xs:complexType>
79   </xs:element>
80   <xs:element name="excludes">
81     <xs:complexType>
82       <xs:attribute name="name" use="required"/>
83       <xs:attribute name="feature" use="required"/>
84       <xs:attribute name="excludes" use="required"/>
85     </xs:complexType>
86   </xs:element>
87
88 </xs:schema>
```

RUNNING A QVT TRANSFORMATION PROGRAMMATICALLY USING MEDINI QVT

Listing F.1: Running a QVT transformation programmatically using medini QVT:*QvtTransformationJob* class

```
1  package es.upv.dsic.issi.qvt.launcher.internal;
2
3  import java.io.IOException;
4  import java.io.InputStreamReader;
5  import java.io.Reader;
6  import java.util.ArrayList;
7  import java.util.Collection;
8  import java.util.Collections;
9  import java.util.HashMap;
10
11 import org.eclipse.core.resources.ResourcesPlugin;
12 import org.eclipse.core.resources.WorkspaceJob;
13 import org.eclipse.core.runtime.CoreException;
14 import org.eclipse.core.runtime.IProgressMonitor;
15 import org.eclipse.core.runtime.IStatus;
16 import org.eclipse.core.runtime.Status;
17 import org.eclipse.emf.common.util.Monitor;
18 import org.eclipse.emf.common.util.URI;
19 import org.eclipse.emf.ecore.EObject;
20 import org.eclipse.emf.ecore.EPackage;
21 import org.eclipse.emf.ecore.resource.Resource;
22 import org.eclipse.emf.ecore.resource.ResourceSet;
23 import org.eclipse.emf.ecore.resource.impl.ResourceSetImpl;
24 import org.eclipse.emf.ecore.xmi.PackageNotFoundException;
25 import org.oslo.ocl20.standard.lib.OclAnyModelElement;
26
27 import traces.TraceabilityLink;
28 import traces.TraceabilityModel;
29 import traces.TracesFactory;
30 import uk.ac.kent.cs.kmf.util.ILog;
31 import uk.ac.kent.cs.kmf.util.OutputStreamLog;
32 import de.ikv.emf.qvt.EMFQvtProcessorImpl;
33 import de.ikv.medini.qvt.QVTProcessorConsts;
34 import de.ikv.medini.qvt.Trace;
35 import es.upv.dsic.issi.qvt.launcher.QvtLauncherPlugin;
```

```
36 import es.upv.dsic.issi.qvt.launcher.model.qvtinvocation.
      Domain;
37 import es.upv.dsic.issi.qvt.launcher.model.qvtinvocation.
      QvtTransformationInvocation;
38
39 public class QvtTransformationJob extends WorkspaceJob {
40
41   QvtTransformationInvocation invocation;
42
43   ResourceSet resourceSet = new ResourceSetImpl();
44
45   public QvtTransformationJob(QvtTransformationInvocation
        invocation) {
46     super("Running " + invocation.getName() + "
          transformation");
47     this.invocation = invocation;
48   }
49
50   @Override
51   public IStatus runInWorkspace(IProgressMonitor monitor)
52       throws CoreException {
53
54     monitor.beginTask("Running...", Monitor.UNKNOWN);
55
56     Resource targetResource = null;
57
58     ILog log = new OutputStreamLog(System.err);
59
60     EMFQvtProcessorImpl emfQvtProcessorImpl = new
          EMFQvtProcessorImpl(log);
61
62     Reader qvtScriptReader = new InputStreamReader(
63          ResourcesPlugin.getWorkspace().getRoot().getFile(
              invocation.getPath()).getContents());
64
65
66     Collection<Resource> models = (new ArrayList<Resource
          >());
67     for (Domain domain : invocation.getDomains()) {
68
69       Resource resource = null;
70
71       if (ResourcesPlugin.getWorkspace().getRoot().getFile
            (domain.getModelPath()).exists()) {
72         try {
73           resource = resourceSet.getResource(
74               URI.createPlatformResourceURI(domain.
                   getModelPath().toString(),false),
                   true);
75           resource.load(Collections.EMPTY_MAP);
```

```
76
77                 if (!resource.getContents().isEmpty()) {
78                    emfQvtProcessorImpl.addMetaModel(resource.
                          getContents().get(0).eClass().
                          getEPackage());
79                 } else {
80                    resource = resourceSet.createResource(
81                          URI.createPlatformResourceURI(domain.
                               getModelPath().toString(),false))
                               ;
82                 }
83              } catch (IOException e) {
84                 return new Status(IStatus.ERROR,
                       QvtLauncherPlugin.PLUGIN_ID,e.
                       getLocalizedMessage(),e);
85              } catch (Exception e) {
86                 if (e.getCause() instanceof
                       PackageNotFoundException) {
87                    return new Status(IStatus.ERROR,
                          QvtLauncherPlugin.PLUGIN_ID,e.getCause
                          ().getLocalizedMessage(),e.getCause());
88                 }
89                 return new Status(IStatus.ERROR,
                       QvtLauncherPlugin.PLUGIN_ID,e.
                       getLocalizedMessage(),e);
90              }
91           } else {
92              resource = resourceSet.createResource(
93                    URI.createPlatformResourceURI(domain.
                          getModelPath().toString(),false));
94
95              resource.getContents().clear();
96
97              Object[] keys = EPackage.Registry.INSTANCE.keySet
                     ().toArray();
98
99
100             for (Object key : keys) {
101                EPackage pkg = EPackage.Registry.INSTANCE.
                       getEPackage(key.toString());
102                if (pkg.getNsPrefix().equals(domain.
                       getNsPrefix())) {
103                   emfQvtProcessorImpl.addMetaModel(pkg);
104                }
105             }
106
107          }
108          models.add(resource);
109       }
110
```

```
111         targetResource = resourceSet.getResource(
112               URI.createPlatformResourceURI(invocation.
                       getDirection().getModelPath().toString(),
                       false),true);
113
114         targetResource.getContents().clear();
115
116         emfQvtProcessorImpl.setModels(models);
117         emfQvtProcessorImpl.setDebug(true);
118
119
120           emfQvtProcessorImpl.setProperty(QVTProcessorConsts.
                   PROP_DISABLE_TRACES, "true");
121
122           //emfQvtProcessorImpl.setProperty(QVTProcessorConsts.
                   PROP_DISABLE_TRANSACTIONAL_MODE, "true");
123
124           try {
125            Collection<Trace> traces = emfQvtProcessorImpl.
                    evaluateQVT(
126                  qvtScriptReader,
127                  invocation.getName(),
128                  true,
129                  invocation.getDirection().getName(),
130                  models.toArray(),
131                  new ArrayList<Trace>(),
132                  log);
133            TraceabilityModel traceabilityModel =
                    createTraceabilityModel(traces);
134
135            Resource tracesResource = resourceSet.createResource
                    (
136                  URI.createPlatformResourceURI(invocation.
                         getDirection().getModelPath().
137                        removeFileExtension().addFileExtension("
                              traces").toString(),false));
138
139            tracesResource.getContents().add(traceabilityModel);
140
141            targetResource.save(Collections.EMPTY_MAP);
142
143            tracesResource.save(Collections.EMPTY_MAP);
144
145           } catch (IOException e) {
146            return new Status(IStatus.ERROR,QvtLauncherPlugin.
                    PLUGIN_ID,e.getLocalizedMessage(),e);
147         } catch (Exception e) {
148            return new Status(IStatus.ERROR,QvtLauncherPlugin.
                    PLUGIN_ID,e.getLocalizedMessage(),e);
149         }
```

```
150
151         return Status.OK_STATUS;
152     }
153
154     private TraceabilityModel createTraceabilityModel(
           Collection<Trace> traces) {
155         TraceabilityModel traceabilityModel = TracesFactory.
               eINSTANCE.createTraceabilityModel();
156
157         traceabilityModel.setName("Transformation");
158
159         for (Domain domain : invocation.getDomains()) {
160            if (domain != invocation.getDirection()) {
161               traceabilityModel.getDomainModels().add(URI.
                     createPlatformResourceURI(domain.getModelPath
                     ().toString(),false));
162            } else {
163               traceabilityModel.getTargetModels().add(URI.
                     createPlatformResourceURI(domain.getModelPath
                     ().toString(),false));
164            }
165         }
166
167
168         for (Trace t : traces) {
169            TraceabilityLink traceabilityLink = TracesFactory.
                  eINSTANCE.createTraceabilityLink();
170
171            traceabilityModel.getLinks().add(traceabilityLink);
172
173            traceabilityLink.setManipulationRule(t.getRelation()
                  .getName());
174
175            for (Object obj1 : t.getBindings()) {
176               for (Object obj2 : ((HashMap)obj1).values())
177               if (obj2 instanceof OclAnyModelElement) {
178                  OclAnyModelElement elt = (OclAnyModelElement)
                        obj2;
179                  if (!((EObject) elt.asJavaObject()).eClass().
                        eResource().equals(
180                        resourceSet.getResource(
181                              URI.createPlatformResourceURI(
                                    invocation.getDirection().
                                    getModelPath().toString(),
                                    false),true)
182                              .getContents().get(0).eClass().
                                    eResource())) {
183                     traceabilityLink.getDomain().add((EObject)
                           elt.asJavaObject());
184                  } else {
```

```
185                         traceabilityLink.getRange().add((EObject)
                                elt.asJavaObject());
186                     }
187                 }
188             }
189         }
190
191         return traceabilityModel;
192     }
193 }
```

G TRANSPATH2CPN TRANSFORMATION

Listing G.1: Full Transpath2CPN transformation

```
transformation TranspathToCPN(tpDomain:transpath, cpnDomain:
      cpn)
{

  // Key declaration

  key Page {name};
  key Globbox {id};
  key Cpnet {page,globbox};
  key Block {idname};
  key Enumerated {idname};
  key Product {idname};
  key ColorSetElement {name};
  key Place {id};
  key Trans {id};
  key Initmark {id};
  key Mark {initmark,colorSetElement};
  key Fusion {name};
  key Arc {trans,place};


  /*************************************************
   ** Root elements
   *************************************************/

  top relation NetworkToCpnet {

    checkonly domain tpDomain network : transpath::Network {
      molecules = molecule : transpath::Molecule {},
      reactions = reaction : transpath::Reaction {}
    };

    enforce domain cpnDomain cpnet : cpn::Cpnet{
      page = page : Page {
        id = network.pathway.id.first(),
        name = network.pathway.name.first(),
        posx = 200,
        posy = 100,
        width = 1000,
```

```
39          height = 800
40        },
41        globbox = globbox : cpn::Globbox {
42          id = 'Declarations'
43        }
44      };
45      where {
46        ComplexMoleculeToProduct(molecule, globbox);
47        ReactionToGUIElements(reaction, page);
48        //page.performLayout(600, 400, 1000);
49      }
50    }
51
52    /**************************************************
53     ** Declarations
54     **************************************************/
55
56    relation ComplexMoleculeToProduct {
57
58      checkonly domain tpDomain molecule : transpath::Molecule
            {
59        statesOf = simpleMolecule : transpath::Molecule {}
60      };
61
62      enforce domain cpnDomain globbox : cpn::Globbox {
63        declarations = resourcesBlock : cpn::Block {
64          id = 'Resources',
65          idname = 'Resources'
66        },
67        declarations = complexesBlock : cpn::Block {
68          id = 'Complexes',
69          idname = 'Complexes',
70          declarations = product : cpn::Product {
71            idname = GetMoleculeType(molecule)
72          }
73        }
74      };
75      when {
76        IsComplexMolecule(molecule);
77      }
78      where {
79        SimpleMoleculeToEnumerated(simpleMolecule,
              resourcesBlock, product);
80      }
81    }
82
83    relation SimpleMoleculeToEnumerated {
84
85      checkonly domain tpDomain molecule : transpath::Molecule
            {
```

```
86      };
87
88      enforce domain cpnDomain resourcesBlock : cpn::Block {
89        declarations = enumerated : cpn::Enumerated {
90          idname = GetMoleculeType(molecule),
91          id = GetMoleculeType(molecule),
92          //usedIn = usedIn->including(product),
93          colorElements = element : cpn::ColorSetElement {
94            name = molecule.name
95          }
96        }
97      };
98      enforce domain cpnDomain product : cpn::Product {
99        simpleColors =
100         simpleColors
101         ->including(enumerated)
102         ->sortedBy(colorSet : ColorSet | colorSet.idname)
103     };
104
105     when {
106       IsSimpleMolecule(molecule);
107     }
108   }
109
110   /*************************************************
111    ** Graphical elements
112    *************************************************/
113
114
115   relation ReactionToGUIElements {
116
117     checkonly domain tpDomain reaction : transpath::Reaction
            {
118       reactantsCoefficient = reactantsCoefficients :
              transpath::ReactantsCoefficient {},
119       producesCoefficient = productsCoefficients : transpath
              ::ProductsCoefficient {}
120     };
121
122     enforce domain cpnDomain page : cpn::Page {
123       transs = transition : cpn::Trans {
124         id = reaction.name,
125         text = 'bind ' + BuildTransitionText(reaction),
126         width = 60,
127         hight = 40,
128         fillColour = 'White',
129         fillPattern = 'Solid',
130         fillFilled = false,
131         lineThick = 1,
132         lineType = 'Solid',
```

```
133         lineColour = 'Black'
134       }
135     };
136
137     where {
138       ReactantToPlaceArc(reactantsCoefficients.reactants,
              transition, reaction.name, page);
139       ProductToPlaceArc(productsCoefficients.produces,
              transition, reaction.name, page);
140     }
141   }
142
143   relation ReactantToPlaceArc {
144
145     checkonly domain tpDomain reactant : Molecule {};
146
147     checkonly domain cpnDomain trans : cpn::Trans {};
148
149     primitive domain reactionName : String;
150
151     enforce domain cpnDomain page : cpn::Page {
152       places = place : cpn::Place {
153         id = GetMoleculeType(reactant),
154         lineColour =
155           if Reaction.allInstances().producesCoefficient.
                  produces->includes(reactant) then
156             'Black'
157           else
158             'Lime'
159           endif
160       },
161       arcs = arc : cpn::Arc {
162         id = '{' + reactant.name + '} => {' + reactionName +
                 '}',
163         orientation = 'PtoT',
164         trans = trans,
165         place = place
166       }
167     };
168     where {
169       FillCommonAttributesInPlaces(place);
170       MoleculeToArcAnnot(reactant,arc);
171       SimpleMoleculeToPlaceType(reactant, place);
172       ComplexMoleculeToPlaceType(reactant, place);
173       SimpleReactantToInitMark(reactant,place);
174     }
175   }
176
177   relation ProductToPlaceArc {
178
```

```
179     checkonly domain tpDomain product : Molecule {};
180
181     checkonly domain cpnDomain trans : cpn::Trans {};
182
183     primitive domain reactionName : String;
184
185     enforce domain cpnDomain page : cpn::Page {
186       places = place : cpn::Place {
187         lineColour =
188           if not(Reaction.allInstances().reactantsCoefficient
                .reactants->includes(product)) then
189             'Maroon'
190           else
191             'Black'
192           endif,
193         id =
194           if Reaction.allInstances().reactantsCoefficient.
                reactants->includes(product) then
195             GetMoleculeType(product)
196           else
197             product.name
198           endif,
199         text =
200           if not(Reaction.allInstances().reactantsCoefficient
                .reactants->includes(product)) then
201             'Dead end'
202           else
203             ''
204           endif
205       },
206       arcs = arc : cpn::Arc {
207         id = '{' + reactionName + '} => {' + product.name + '
              }',
208         orientation = 'TtoP',
209         trans = trans,
210         place = place
211       }
212     };
213
214     where {
215       FillCommonAttributesInPlaces(place);
216       MoleculeToArcAnnot(product,arc);
217       SimpleMoleculeToPlaceType(product, place);
218       ComplexMoleculeToPlaceType(product, place);
219     }
220   }
221
222   relation FillCommonAttributesInPlaces {
223
224     enforce domain cpnDomain place : cpn::Place {
```

```
225         width = 60,
226         hight = 40,
227         fillColour = 'White',
228         fillPattern = 'Solid',
229         fillFilled = false,
230         lineThick = 1,
231         lineType = 'Solid'
232     };
233
234   }
235
236   relation SimpleReactantToInitMark {
237
238     checkonly domain tpDomain reactant : transpath::Molecule
          {
239     };
240
241     enforce domain cpnDomain place : cpn::Place  {
242       initmark = imark : cpn::Initmark {
243         fillColour = 'White',
244         fillPattern = 'Solid',
245         fillFilled = false,
246         lineThick = 1,
247         lineType = 'Solid',
248         lineColour = 'Lime',
249
250         id = GetMoleculeType(reactant),
251         marks = mark : cpn::Mark {
252           value =
253             --- This value is not a valid value, it is set in
                  this way in order to
254             --- obtain a valid CPNet. This marking must be
                 corrected by biologists.
255             --- Now, we initialize this value to the number
                 of molecules of each kind
256             --- that are involved in the reactions of the
                 pathway
257             reactant.rkoutsCoefficient.coefficient->sum(),
258           colorSetElement = colorSetElement : cpn::
                ColorSetElement {
259             name = reactant.name
260           }
261         }
262       }
263     };
264
265     when {
266       IsSimpleMolecule(reactant);
267     }
268   }
```

```
269
270  relation SimpleMoleculeToPlaceType {
271
272    checkonly domain tpDomain molecule : Molecule {};
273
274    enforce domain cpnDomain place : Place {
275      type = enumerated : cpn::Enumerated {
276        idname = GetMoleculeType(molecule)
277      }
278    };
279    when {
280      IsSimpleMolecule(molecule);
281    }
282  }
283
284  relation ComplexMoleculeToPlaceType {
285
286    checkonly domain tpDomain molecule : Molecule {};
287
288    enforce domain cpnDomain place : Place {
289      type = product : cpn::Product {
290        idname = GetMoleculeType(molecule)
291      }
292    };
293    when {
294      IsComplexMolecule(molecule);
295    }
296  }
297
298  relation MoleculeToArcAnnot {
299
300    checkonly domain tpDomain molecule : Molecule {};
301
302    enforce domain cpnDomain arc : cpn::Arc {
303      headsize = 1,
304      currentcyckle = '2',
305      fillColour = 'White',
306      fillPattern = 'Solid',
307      fillFilled = false,
308      lineThick = 1,
309      lineType = 'Solid',
310      lineColour =
311        if IsSimpleMolecule(molecule) then
312          'Lime'
313        else
314          'Black'
315        endif,
316      annot = annot : cpn::Annot {
317        fillColour = 'White',
318        fillPattern = 'Solid',
```

```
319         fillFilled = false,
320         lineThick = 1,
321         lineType = 'Solid',
322         lineColour =
323         if IsSimpleMolecule(molecule) then
324           'Lime'
325         else
326           'Black'
327         endif,
328         text =
329         if IsSimpleMolecule(molecule) then
330           '1`' + molecule.name
331         else
332           '(' + BuildMoleculeComponentsList(molecule) + ')'
333         endif
334       }
335     };
336
337   }
338
339   /************************************************
340    ** Helper functions
341    ************************************************/
342
343
344   query IsSimpleMolecule(molec:Molecule):Boolean
345   {
346     (molec.statesOf -> isEmpty())
347   }
348
349   query IsComplexMolecule(molec:Molecule):Boolean
350   {
351     (not(molec.statesOf -> isEmpty()))
352   }
353
354   query GetMoleculeType(molecule : transpath::Molecule) :
        String
355   {
356     if IsSimpleMolecule(molecule) then
357       GetSimpleMoleculeType(molecule)
358     else
359       GetComplexMoleculeType(molecule)
360     endif
361   }
362
363   query GetSimpleMoleculeType(molec : Molecule) : String
364   {
365     if (molec.klass -> includes('adaptor proteins'))
366     then 'A'
367     else
```

```
368       if (molec.klass -> includes('receptors'))
369       then 'R'
370       else 'O'
371       endif
372     endif
373   }
374
375   query GetComplexMoleculeType(complexMolecule : Molecule) :
          String
376   {
377     complexMolecule.statesOf
378     ->collect(m : Molecule | GetSimpleMoleculeType(m))
379     ->asSet()
380     ->sortedBy(type : String | type )
381     ->iterate(type : String; complexBlockID : String = '' |
            complexBlockID.concat(type))
382   }
383
384   query BuildTransitionText(reaction : Reaction) : String
385   {
386     let moleculeNames : Sequence(String) =
387       reaction.reactantsCoefficient.reactants.name
388     in
389       moleculeNames
390       ->excluding(moleculeNames.last())
391       ->iterate(moleculeName : String; text : String = '' |
            text.concat(moleculeName).concat(','))
392       .concat(moleculeNames.last())
393   }
394   query BuildMoleculeComponentsList(molecule : Molecule) :
          String
395   {
396
397     let moleculesSet : OrderedSet(Molecule) =
398       molecule.statesOf
399       ->asSet()
400       ->sortedBy(molecule : Molecule | GetMoleculeType(
            molecule))
401     in
402       moleculesSet
403       ->excluding(moleculesSet.last())
404       ->iterate(molecule : Molecule; text : String = '' |
            text.concat(molecule.name).concat(','))
405       .concat(moleculesSet.last().name)
406   }
407 }
```

BIBLIOGRAPHY

Airbus Société par Actions Simplifiée (2011). *Airbus S.A.S website*. URL: `http://www.airbus.com/` (cited on page 277).

Alberts, B., D. Bray, K. Hopkin, A. Johnson, J. Lewis, M. Raff, K. Roberts and P. Walter (2005). *Lehrbuch der Molekularen Zellbiologie*. 3rd edition. Weinheim: Wiley-VCH Verlag GmbH / Co. KGaA, page 908. ISBN: 978-3-527-31160-6 (cited on page 311).

Alexander, C. (1977). *A pattern language : towns, buildings, construction*. New York: Oxford University Press. ISBN: 9780195019193 (cited on page 250).

Ali, N. (2008). Ambient-PRISMA: ambients in aspect-oriented software architectures. PhD thesis. Universidad Politécnica de Valencia (cited on page 70).

AMPLE (2011). *European Commission STREP Project AMPLE IST-033710*. URL: `http://www.ample-project.net` (cited on page 374).

Antkiewicz, M. and K. Czarnecki (2004). FeaturePlugin: feature modeling plug-in for Eclipse. In: *2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72 (cited on page 370).

Apache Foundation (2011). *Apache Commons Collections API*. URL: `http://commons.apache.org/collections/` (cited on page 167).

Appukuttan, B., T. Clark, L. Tratt, S. Reddy, R. Venkatesh, A. Evans, G. Maskeri, P. Sammut and J. Willans (2003). *Revised submission for MOF 2.0 Query / Views / Transformations RFP*. Technical report ver. 1.1. QVT-Partners. URL: `http://eprints.bournemouth.ac.uk/5974/` (cited on page 41).

Apt, K. R. and M. Wallace (2007). *Constraint Logic Programming using Eclipse*. New York, NY, USA: Cambridge University Press. ISBN: 0521866286 (cited on page 373).

AtlanMod Main Page (2011). URL: `http://www.emn.fr/z-info/atlanmod/` (cited on page 41).

ATLAS INRIA & LINA (2005). *ATL: the ATLAS Transformation Language*. URL: http://atlanmod.emn.fr/www/papers/ATL/ATL_Documentation/ATL_PresentationSheet.pdf (cited on page 41).

Auer, M., B. Graser and S. Biffl (2003). A Survey on the Fitness of Commercial Software Metric Tools for Service in Heterogeneous Environments: Common Pitfalls. In: *IEEE METRICS'03*, pages 144–144 (cited on page 335).

Avila-García, O., A. Estévez and J. L. Roda (2006). Integrando modelos de procesos y activos reutilizables en una herramienta MDA. In: *JISBD'06 proceedings*, pages 483–488. ISBN: 84-95999-99-4 (cited on page 374).

Bachmann, F., M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh and A. Vilbig (2004). A Meta-model for Representing Variability in Product Family Development. In: *Software Product-Family Engineering*, pages 66–80. URL: http://www.springerlink.com/content/4xfd43n76rm0nwvt (cited on page 374).

Bass, L., P. Clements and R. Kazman (1998). *Software architecture in practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-19930-0 (cited on pages 67 sq., 70, 146).

Batory, D., D. Benavides and A. Ruiz-Cortés (2006). Automated Analysis of Feature Models: Challenges Ahead. In: *Communications of the ACM* December (cited on page 373).

Batory, D. (2005). Feature models, grammars, and propositional formulas. In: Springer, pages 7–20 (cited on pages 53, 58, 372).

Batory, D. S. (2003). A Tutorial on Feature Oriented Programming and Product-Lines. In: *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*. IEEE Computer Society, pages 753–754 (cited on page 273).

Benavides, D., S. Segura and A. Ruiz-Cortés (2010). Automated analysis of feature models 20 years later: A literature review. In: *Information Systems* 35.6, pages 615 –636. ISSN: 0306-4379. DOI: DOI:10.1016/j.is.2010.01.001. URL: http://www.sciencedirect.com/science/article/pii/S0306437910000025 (cited on page 274).

Beuche, D. (2007). Modeling and Building Software Product Lines with pure: : variants. In: *SPLC (2)*. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, pages 143–144. ISBN: 978-4-7649-0342-5 (cited on page 371).

Bézivin, J. (2004). In Search of a Basic Principle for Model Driven Engineering. In: *UPGRADE – The European Journal for the Informatics Professional* 5.2, pages 21–24 (cited on page 349).

Bézivin, J., F. Jouault and D. Touzet (2005). Principles, Standards and Tools for Model Engineering. In: *ICECCS*. IEEE Computer Society, pages 28–29. ISBN: 0-7695-2284-X (cited on page 334).

Bhattacharya, K., R. Guttman, K. Lyman, F. Heath, S. Kumaran, P. Nandi, F. Wu, P. Athma, C. Freiberg, L. Johannsen, et al. (2005). A model-driven approach to industrializing discovery processes in pharmaceutical research. In: *IBM Systems Journal* 44.1, pages 145–162 (cited on page 330).

Boeing Company (2011). *Boeing Company website*. URL: `http://www.boeing.com/` (cited on page 277).

Boronat, A. (2007). MOMENT: a formal framework for MOdel manageMENT. PhD thesis. Universitat Politènica de València (UPV), Spain. URL: `http://hdl.handle.net/10251/1964` (cited on page 41).

Boronat, A., J. A. Carsí and I. Ramos (2005a). Automatic Support for Traceability in a Generic Model Management Framework. In: *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005*. Edited by A. Hartman and D. Kreische. Volume 3748. Lecture Notes in Computer Science. Springer, pages 316–330. ISBN: 3-540-30026-0 (cited on pages 171, 173).

Boronat, A., J. Iborra, J. Ángel Carsí, I. Ramos and A. Gómez (2005b). Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework. In: *Revista IEEE América Latina* (cited on page 41).

Boronat, A., J. Iborra, J. A. Carsí, I. Ramos and A. Gómez (2005c). Del método formal a la aplicación industrial en Gestión de Modelos: Maude aplicado a Eclipse Modeling Framework. In: *Actas X*

Jornadas sobre Ingeniería del Software y Bases de Datos. JISBD'05. Granada, Spain. eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=26&Itemid=28 (cited on page 385).

Boronat, A., J. Iborra, J. A. Carsí, I. Ramos and A. Gómez (2005d). Utilización de Maude desde Eclipse Modeling Framework para la Gestión de Modelos. In: *Desarrollo de Software Dirigido por Modelos - DSDM'05 (Junto a JISBD'05). Granada, Spain.* eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=32 (cited on page 385).

Boronat, A., J. Oriente, A. Gómez, I. Ramos and J. A. Carsí (2006a). An Algebraic Specification of Generic OCL Queries. In: *Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006.* Lecture Notes in Computer Science 4066, pages 316–330. eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=22&Itemid=28. URL: http://dx.doi.org/10.1007/11787044_24 (cited on page 386).

Boronat, A., J. Oriente, A. Gómez, I. Ramos and J. A. Carsí (2006b). MOMENT-OCL: Algebraic Specifications of OCL 2.0 within the Eclipse Modeling Framework. In: *ENTCS. Demo. 6th International Workshop on Rewriting Logic and its Applications. Vienna, Austria.* eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=42 (cited on page 386).

Boronat, A., A. Knapp, J. Meseguer and M. Wirsing (2008). What Is a Multi-modeling Language? In: *WADT*. Edited by A. Corradini and U. Montanari. Volume 5486. Lecture Notes in Computer Science. Springer, pages 71–87. ISBN: 978-3-642-03428-2 (cited on page 66).

Bruneliere, H., J. Cabot, F. Jouault and F. Madiot (2010). MoDisco: a generic and extensible framework for model driven reverse engineering. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering.* ASE '10. Antwerp, Belgium: ACM, pages 173–174. ISBN: 978-1-4503-0116-9. DOI: http://doi.acm.org/10.1145/1858996.1859032. URL: http://doi.acm.org/10.1145/1858996.1859032 (cited on page 40).

Bryant, R. (1986). Graph-Based Algorithms for Boolean Function Manipulation. In: *Computers, IEEE Transactions on* C-35.8, pages 677 –691. ISSN: 0018-9340. DOI: 10.1109/TC.1986.1676819 (cited on page 274).

Burbeck, S. (1987). *Applications Programming in Smalltalk-80: How to use Model-View- Controller (MVC)*. Palo Alto, CA: Softsmarts, Inc. URL: http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html (cited on page 38).

Cabello, M. E. and I. Ramos (2009). Expert Systems Development Through Software Product Lines Techniques. In: *Information Systems Development*. Springer US, pages 299–307. ISBN: 978-0-387-84809-9 (Print) 978-0-387-84810-5 (Online). DOI: 10.1007/b137171_31. URL: http://www.springerlink.com/content/g7465u5p72467153/ (cited on pages 226, 273).

Cabello, M. E., I. Ramos, A. Gómez and R. Limón (2009). Baseline-Oriented Modeling: An MDA Approach Based on Software Product Lines for the Expert Systems Development. In: *Intelligent Information and Database Systems, Asian Conference on* 0, pages 208–213. DOI: http://doi.ieeecomputersociety.org/10.1109/ACIIDS.2009.15 (cited on pages 273, 387).

Cabello, M. E., I. Ramos, J. R. Gutiérrez, A. Gómez and r. Limon (2011). SPL variability management, cardinality and types: an MDA approach. In: *International Journal of Intelligent Information and Database Systems*. Unpublished yet, in press (cited on page 389).

Cabello Espinosa, M. E. (2008). Baseline-oriented modeling: una aproximación MDA basada en líneas de productos software para el desarrollo de aplicaciones: (dominio de los sistemas expertos). PhD thesis. Universidad Politécnica de Valencia. URL: http://hdl.handle.net/10251/3793 (cited on pages 115, 220, 222, 229).

Cabot, J., R. Clariso and D. Riera (2008). Verification of UML/OCL Class Diagrams using Constraint Programming. In: *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, pages 73 –80. DOI: 10.1109/ICSTW.2008.54 (cited on page 373).

Cabot, J., R. Clarisó and D. Riera (2007). UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ASE '07. Atlanta, Georgia, USA: ACM, pages 547–548. ISBN: 978-1-59593-882-4. DOI: http://doi.acm.org/10.1145/1321631.1321737. URL: http://doi.acm.org/10.1145/1321631.1321737 (cited on page 373).

Chen, L., M. A. Babar and N. Ali (2009). Variability Management in Software Product Lines: A Systematic Review. In: *Proceedings of the 13th International Software Product Lines Conference (SPLC'09), San Francisco, CA, USA*. URL: http://www.sei.cmu.edu/splc2009/ (cited on pages 50 sq., 370).

Chung, L., B. A. Nixon, E. Yu and J. Mylopoulos (2000). *Non-functional requirements in software engineering*. English. Kluwer International Series in Software Engineering 5. Boston, MA: Kluwer Academic Publishers. (cited on page 351).

Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. F. Quesada (2002). Maude: specification and programming in rewriting logic. In: *Theor. Comput. Sci.* 285.2, pages 187–243 (cited on page 41).

Clements, P. and L. Northrop (2001). *Software Product Lines: practices and patterns*. Volume 0201703327. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. (cited on pages 45 sq., 48, 373).

Clements, P., F. Bachman, L. Bass, D. Garland, J. Ivers, R. Little, P. Merson, R. Nord and J. Stafford (2003). *Documenting software architectures: views and beyond*. SEI series in software engineering. Addison-Wesley. ISBN: 9780201703726 (cited on pages 68, 70).

Cook, S. and D. Mitchell (1997). Satisfiability Problem: Theory and Applications. In: *Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society (cited on page 274).

Cook, S., G. Jones, S. Kent and A. Wills (2007). *Domain-specific development with visual studio dsl tools*. 1st edition. Addison-Wesley Professional. ISBN: 9780321398208 (cited on page 363).

Costa-Soria, C. (2011). Dynamic Evolution and Reconfiguration of Software Architectures through Aspects. PhD thesis. Universidad Politécnica de Valencia. URL: http://hdl.handle.net/10251/11038 (cited on pages 70, 160 sq.).

Courtois, P. J. (1985). On time and space decomposition of complex structures. In: *Commun. ACM* 28 (6), pages 590–603. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/3812.3814. URL: http://doi.acm.org/10.1145/3812.3814 (cited on page 66).

Czarnecki, K. and S. Helsen (2006). Feature-based survey of model transformation approaches. In: *IBM Systems Journal* 45.3, pages 621 –645. ISSN: 0018-8670. DOI: 10.1147/sj.453.0621 (cited on page 352).

Czarnecki, K. and M. Antkiewicz (2005). Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: *GPCE*. Edited by R. Glück and M. R. Lowry. Volume 3676. Lecture Notes in Computer Science. Springer, pages 422–437. ISBN: 3-540-29138-5 (cited on page 273).

Czarnecki, K. and U. W. Eisenecker (2000). *Generative programming: methods, tools, and applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co. ISBN: 0-201-30977-7 (cited on pages 15, 51, 273, 310).

Czarnecki, K. and C. H. Kim (2005). *Cardinality-Based Feature Modeling and Constraints: A Progress Report*. Technical report. 200 University Ave. West Waterloo, ON N2L 3G1, Canada: University of Waterloo (cited on pages 58, 370 sq.).

Czarnecki, K., S. Helsen and E. Ulrich (2005a). Formalizing cardinality-based feature models and their specialization. In: *Software Process: Improvement and Practice* 10, pages 7 –29. ISSN: 1099-1670. DOI: 10.1002/spip.213. URL: http://www3.interscience.wiley.com/cgi-bin/fulltext/109932076/PDFSTART (cited on pages 56 sq., 75).

Czarnecki, K., S. Helsen and U. Eisenecker (2005b). Staged configuration through specialization and multilevel configuration of feature models. In: *Software Process: Improvement and Prac-*

tice 10.2, pages 143–169. DOI: 10 . 1002 / spip . 225. URL: http : //dx.doi.org/10.1002/spip.225 (cited on page 75).

Dahl, O. J., E. W. Dijkstra and C. A. R. Hoare, editors (1972). *Structured programming*. London, UK, UK: Academic Press Ltd. ISBN: 0-12-200550-3 (cited on page 13).

Damm, W. and D. Harel (2001). LSCs: Breathing Life into Message Sequence Charts. In: *Formal Methods in System Design* 19.1, pages 45–80 (cited on page 315).

Dardenne, A., A. van Lamsweerde and S. Fickas (1993). Goal-directed requirements acquisition. In: *Science of Computer Programming* 20.1-2, pages 3 –50. ISSN: 0167-6423. DOI: 10.1016/0167-6423(93)90021-G. URL: http://www.sciencedirect.com/science/article/pii/016764239390021G (cited on page 351).

Dauphinee, S. M. and A. Karsan (2006). Lipopolysaccharide signaling in endothelial cells. In: *Lab Invest* 86.1, pages 9–22 (cited on page 314).

Del Fabro, M. D. and F. Jouault (2005). Model Transformation and Weaving in the AMMA Platform. In: *Generative and Transformational Techniques in Software Engineering (GTTSE), Workshop*, pages 71–77. URL: http://www.lina.sciences.univ-nantes.fr/Publications/2005/DJ05 (cited on page 173).

Demathieu, S., C. Griffin and S. Sendall (2005). Model Transformation with the IBM Model Transformation Framework. In: URL: http://www.ibm.com/developerworks/rational/library/05/503_sebas/ (cited on pages 41 sq.).

Dennis, M. B. and D. Goldenson (2004). Measurement and Analysis: What Can and Does Go Wrong? In: *in: Proceedings of the 10th International Symposium on Software Metrics (METRICS'04*, pages 131–138 (cited on pages 333, 335).

Duran-Limon, H. A., F. E. Castillo-Barrera and R. E. Lopez-Herrejon (2011). Towards an ontology-based approach for deriving product architectures. In: *Proceedings of the 15th International Software Product Line Conference, Volume 2*. SPLC '11. Munich, Germany: ACM, 29:1–29:5. ISBN: 978-1-4503-0789-5. DOI: http://doi.acm.

org/10.1145/2019136.2019169. URL: http://doi.acm.org/10.1145/2019136.2019169 (cited on page 371).

Eclipse Foundation (2003). *Eclipse Platform Technical Overview*. Object Technology International, Inc. URL: http://www.eclipse.org/whitepapers/eclipse-overview.pdf (cited on page 32).

Eclipse Foundation (2011a). *Eclipse Modeling Framework Project (EMF)*. URL: http://www.eclipse.org/modeling/emf/ (cited on page 5).

Eclipse Foundation (2011b). *Model Development Tools (MDT)*. URL: http://www.eclipse.org/modeling/mdt/ (cited on pages 40 sq.).

Eclipse Foundation (2011c). *Papyrus subproject*. URL: http://www.eclipse.org/modeling/mdt/papyrus/ (cited on page 40).

Eclipse Foundation (2011d). *Standard Widget Toolkit (SWT)*. URL: http://www.eclipse.org/swt/ (cited on page 182).

Eclipse Foundation (2011e). *The Graphical Modeling Framework*. URL: http://www.eclipse.org/gmf/ (cited on pages 38, 363).

EFTCoR European Union project (2002-2005). *GROWTH G3RD-CT-00794. EFTCoR: Environmental Friendly and Cost-Effective Technology for Coating Removal. European Project, 5th Framework Prog.* (Cited on page 352).

Emmett, S., S. Rison, S. Abiteboul, et al. (2006). *Towards 2020 Science*. Technical report. Microsoft Corporation. URL: http://research.microsoft.com/towards2020science/downloads/T2020S_ReportA4.pdf (cited on page 309).

Eriksson, M., H. Morast, J. Börstler and K. Borg (2005). The PLUSS toolkit - extending telelogic DOORS and IBM-rational rose to support product line use case modeling. In: *ASE*. Edited by D. F. Redmiles, T. Ellman and A. Zisman. ACM, pages 300–304 (cited on page 55).

Federal Aviation Administration (2007). *Airplane Flying Handbook*. Edited by Federal Aviation Administration and U.S. Dept. of Transportation. Skyhorse Publishing. ISBN: 9781602390034. URL: http://books.google.com/books?id=V3SZXFWuCIgC (cited on page 276).

Fenton, N. E. and S. L. Pfleeger (1998). *Software Metrics: A Rigorous and Practical Approach*. Boston, MA, USA: PWS Publishing Co. ISBN: 0534954251. URL: http://portal.acm.org/citation.cfm?id=580949 (cited on page 333).

Filman, R. E., T. Elrad, S. Clarke and M. Akşit (2005). *Aspect-Oriented Software Development*. Boston: Addison-Wesley. ISBN: 0-321-21976-7 (cited on pages 161, 375).

Fisher, J., D. Harel, E. Hubbard, N. Piterman, M. Stern and N. Swerdlin (2004). Combining State-Based and Scenario-Based Approaches in Modeling Biological Systems. In: LNBI, vol. 3082, pages 236–241. ISBN: 978-3-540-25375-4 (cited on page 315).

Florac, W. A., A. D. Carleton and J. R. Barnard (2000). Statistical Process Control: Analyzing a Space Shuttle Onboard Software Process. In: *IEEE Softw.* 17 (4), pages 97–106. ISSN: 0740-7459. DOI: 10.1109/52.854075. URL: http://dl.acm.org/citation.cfm?id=624638.626160 (cited on page 333).

Free Software Foundation (2011). *The Free Software Definition*. URL: http://www.gnu.org/philosophy/free-sw.html (cited on page 42).

Fuentes, L., C. Nebrera and P. Sánchez (2009). Feature-Oriented Model-Driven Software Product Lines: The TENTE Approach. In: *Proceedings of the Forum at the CAiSE 2009 Conference*. Edited by E. Yu, J. Eder and C. Rolland. Volume 453. CEUR Workshop Proceedings. Amsterdam (The Netherlands), pages 67–72 (cited on page 374).

Gamma, E., R. Helm, R. E. Johnson and J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley. ISBN: 978-0-201-63361-0 (cited on pages 35, 74).

Gao, S., C. M. Sperberg-McQueen, H. S. Thompson, N. Mendelsohn, D. Beech and M. Maloney (2008). *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. World Wide Web Consortium, Working Draft WD-xmlschema11-1-20080620 (cited on page 40).

García, F., M. F. Bertoa, C. Calero, A. Vallecillo, F. Ruíz, M. Piattini and M. G. A (2005). Towards a consistent terminology for software measurement. In: *Information and Software Technology* 48, pages 631–644 (cited on pages 334, 337 sq.).

García, F., M. Piattini, F. Ruiz, G. Canfora and C. A. Visaggio (2006). FMESP: framework for the modeling and evaluation of software processes. In: *J. Syst. Archit.* 52 (11), pages 627–639. ISSN: 1383-7621. DOI: 10.1016/j.sysarc.2006.06.007. URL: http://dl.acm.org/citation.cfm?id=1226446.1226450 (cited on pages 334, 337).

Garcia, F., M. Serrano, J. Cruz-Lemus, F. Ruiz and M. Piattini (2007). Managing software process measurement: A metamodel-based approach. In: *Inf. Sci.* 177.12, pages 2570–2586. ISSN: 0020-0255. DOI: 10.1016/j.ins.2007.01.018 (cited on pages 334, 337).

Garwood, K., C. Garwood, C. Hedeler, T. Griffiths, N. Swainston, S. Oliver and N. Paton (2006). Model-driven user interfaces for bioinformatics data resources: regenerating the wheel as an alternative to reinventing it. In: *BMC Bioinformatics* 7.1, page 532 (cited on page 330).

GE Aviation (2011). *GE Aviation website*. URL: http://www.geaviation.com/ (cited on page 276).

Giarratano, J. C. and G. Riley (Oct. 2005). *Expert systems : principles and programming*. Thomson Course Technology, c2005. ISBN: 0534384471. URL: http://www.worldcat.org/isbn/0534384471 (cited on pages 219, 222).

Gómez, A., A. Boronat, J. Carsí and I. Ramos (2008). Biological Data Migration in Pathway Simulation. In: *Actas de las VIII Jornadas Nacionales de Bioinformática (JNB'08), Valencia.* (Cited on page 390).

Gómez, A. (2005). Proyecto Final de Carrera: Soporte Gráfico para Trazabilidad en una Herramienta de Gestión de Modelos. Master's thesis. Universidad Politéncica de Valencia (cited on page 171).

Gómez, A. (2008). *Trabajo de investigación: Recuperación, transformación y simulación de datos biológicos mediante ingeniería dirigida por modelos*. Technical report. Universidad Politécnica de Valencia. URL: http://issi.dsic.upv.es/~agomez/bioinforma

tics/Memoria-trabajos-investigacion-agomez.pdf (cited on pages 170, 179, 384).

Gómez, A. and I. Ramos (2010). Automatic Tool Support for Cardinality-Based Feature Modeling with Model Constraints for Information Systems Development. In: *19th International Conference on Information Systems Development, Prague, Czech Republic, August 25 - 27, 2010*. eprint: http://issi.dsic.upv.es/publications/archives/f-1285082432517/document.pdf (cited on page 388).

Gómez, A., A. Boronat, J. A. Carsí and I. Ramos (2005). Integración de un sistema de reescritura de términos en una herramienta de desarrollo software industrial. In: *IV Jornadas de trabajo DYNAMICA. Archena (Murcia)*. eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=31 (cited on page 386).

Gómez, A., A. Boronat, L. Hoyos, J. . Carsí and I. Ramos (2006). Definición de operaciones complejas con un lenguaje específico de dominio en Gestión de Modelos. In: *XI Jornadas de Ingeniería del Software y Bases de Datos. JISBD'06. Sitges, Barcelona. Spain.* Edited by J. C. Riquelme and P. Botella. eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=30 (cited on page 387).

Gómez, A., J. A. Carsí, A. Boronat, I. Ramos, C. Täubner and S. Eckstein (2007a). Biological Data Migration Using a Model Driven Approach. In: *4th International Workshop on Software Language Engineering (ateM 2007). Within ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007). Nashville, TN, USA.* Edited by J.-M. Favre, D. Gasevic, R. Lämmel and A. Winter. ISSN: 0931-9972. eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=39&Itemid=28 (cited on page 389).

Gómez, A., A. Boronat, J. A. Carsí and I. Ramos (2007b). MOMENT-CASE: Un prototipo de herramineta CASE. In: *XII Jornadas de Ingeniería del Software y Bases de Datos. JISBD07. Zaragoza. Spain.* Edited by X. Franch. eprint: http://moment.dsic.upv.es/index.

php?option=com_docman&task=doc_download&gid=40&Itemid=28 (cited on page 386).

Gómez, A., A. Boronat, J. A. Carsí, I. Ramos, C. Täubner and S. Eckstein (2007c). Recuperación y procesado de datos biológicos mediante Ingeniería Dirigida por Modelos. In: *XII Jornadas de Ingeniería del Software y Bases de Datos. JISBD07. Zaragoza. Spain.* Edited by X. Franch. eprint: http://moment.dsic.upv.es/index.php?option=com_docman&task=doc_download&gid=41&Itemid=28. URL: http://www.sistedes.es/sistedes/pdf/2007/JISBD-07-gomez-biologicos.pdf (cited on page 389).

Gómez, A., M. E. Cabello and I. Ramos (2010). BOM-Lazy: A Variability-Driven Framework for Software Applications Production Using Model Transformation Techniques. In: *2nd International Workshop on Model-driven Approaches in Software Product Line Engineering (MAPLE 2010), collocated with the 14th International Software Product Line Conference (SPLC 2010). South Korea, September 2010*. eprint: http://issi.dsic.upv.es/publications/archives/f-1286208702244/document(final).pdf (cited on page 388).

Gómez Lacruz, M., A. Gómez Llana, M. E. Cabello Espinosa and I. Ramos Salavert (2009). BOM-Lazy: gestión de la variabilidad en el desarrollo de Sistemas Expertos mediante técnicas de MDA. In: *VI Taller sobre Desarrollo de Software Dirigido por Modelos (DSDM'09). Junto a XIV Jornadas de Ingeniería del Software y Bases de Datos. 8 de Septiembre de 2009, San Sebastián, España.* eprint: http://issi.dsic.upv.es/publications/archives/f-1249040813544/dsdm_camera_ready.pdf (cited on page 388).

Gómez Llana, A. and I. Ramos Salavert (2010). Cardinality-based feature models and Model-Driven Engineering: fitting them together. In: *Proceedings of the Fourth International Workshop on Variability Modelling of Software-Intensive Systems. ICB-research report No. 37. Pages 61–68. ISSN 1860 - 2770 (Print), ISSN 1866 - 5101 (Online).* eprint: http://issi.dsic.upv.es/publications/archives/f-1263570538890/document.pdf. URL:

http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf (cited on page 388).

Gomez, A., A. Boronat, J. A. Carsi, I. Ramos, C. Taubner and S. Eckstein (2008). JISBD2007-03: Biological Data Processing using Model Driven Engineering. In: *Latin America Transactions, IEEE (Revista IEEE America Latina)* 6.4, pages 324–331. ISSN: 1548-0992. DOI: 10.1109/TLA.2008.4815285 (cited on page 390).

Gong, M., L. Scott, Y. Xiao and R. Offen (1997). A rapid development model for meta-CASE tool design. In: *Conceptual Modeling – ER '97*. Edited by D. Embley and R. Goldstein. Volume 1331. Lecture Notes in Computer Science. 10.1007/3-540-63699-4_37. Springer Berlin / Heidelberg, pages 464–477. ISBN: 978-3-540-63699-1. URL: http://dx.doi.org/10.1007/3-540-63699-4_37 (cited on page 364).

González, A. S., D. S. Ruiz and G. M. Perez (2010). EMF4CPP: a C++ Ecore Implementation. In: Valencia, Spain. eprint: http://www.sistedes.es/TJISBD/Vol-4/No-2/articles/DSDM-articulo-13.pdf. URL: http://www.sistedes.es/TJISBD/Vol-4/No-2/articles/DSDM-articulo-13.pdf (cited on page 37).

González-Huerta, J. (2010). Transformación de modelos dirigida por atributos de calidad. spa. Degree's thesis. Universidad Politécnica de Valencia. URL: http://hdl.handle.net/10251/8627 (cited on page 382).

González-Huerta, J. (2011). Integration of Quality Attributes in Software Product Line Development. spa. Master's thesis. Universidad Politécnica de Valencia (cited on page 382).

Griss, M. L., J. Favaro and M. d'Alessandro (1998). Integrating Feature Modeling with the RSEB. In: *In Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85 (cited on page 54).

Groher, I. and M. Voelter (2009). Aspect-Oriented Model-Driven Software Product Line Engineering. In: *Transactions on Aspect-Oriented Software Development VI*. Edited by S. Katz, H. Ossher, R. France and J.-M. Jézéquel. Volume 5560. Lecture Notes in Com-

puter Science. Springer Berlin / Heidelberg, pages 111–152. ISBN: 978-3-642-03763-4. URL: http://dx.doi.org/10.1007/978-3-642-03764-1_4 (cited on pages 374 sqq.).

Harrison, W. (2004). A flexible method for maintaining software metrics data: a universal metrics repository. In: *Journal of Systems and Software* 72.2, pages 225–234 (cited on page 335).

Heidenreich, F., J. Kopcsek and C. Wende (May 2008). FeatureMapper: Mapping Features to Models. In: *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. Leipzig, Germany: ACM, pages 943–944. ISBN: 978-1-60558-079-1. DOI: http://doi.acm.org/10.1145/1370175.1370199 (cited on page 371).

Hofmeister, C., R. L. Nord and D. Soni (1999). Describing Software Architecture with UML. In: *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*. WICSA1. Deventer, The Netherlands, The Netherlands: Kluwer, B.V., pages 145–160. ISBN: 0-7923-8453-9. URL: http://dl.acm.org/citation.cfm?id=646545.696368 (cited on page 68).

Hucka, M. et al. (2004). Evolving a Lingua Franca and Associated Software Infrastructure for Computational Systems Biology: The Systems Biology Markup Language (SBML) Project. In: *Systems Biology* 1.1, pages 41–53 (cited on page 313).

Hudson, S. E. (2011). *CUP Parser Generator for Java*. URL: http://www.cs.princeton.edu/~appel/modern/java/CUP/ (cited on page 167).

IBM (2011). *IBM Rational DOORS*. URL: http://www-01.ibm.com/software/awdtools/doors/productline/ (cited on page 279).

ikv++ technologies AG (2011). *ikv++ technologies AG website*. URL: http://www.ikv.de (cited on pages 42, 166).

INRIA and LINA (2011). *ATLAS Transformation Language*. URL: http://www.eclipse.org/atl/ (cited on page 41).

Institute of Electrical and Electronics Engineers (1998). *IEEE standard for software test documentation*. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=741968 (cited on page 51).

ISA Research Group (2011a). *FaMa Tool Suite*. University of Seville. URL: http://www.isa.us.es/fama/ (cited on page 274).

ISA Research Group (2011b). *FaMa user's guide*. URL: http://famats.googlecode.com/files/FaMa%20User%20Manual.pdf (cited on page 207).

ISO/IEC JTC1/SC7 N4098 (2008). *ISO/IEC CD 25010: Software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Software and quality in use models* (cited on page 351).

Jacobson, I., M. Griss and P. Jonsson (1997). *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley Professional. ISBN: 978-0-201-92476-3 (cited on page 54).

Jensen, K. (1997). *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*. 2nd edition. Berlin: Springer. ISBN: 978-3-540-60943-8 (cited on page 315).

Jensen, K., L. Kristensen and L. Wells (2007). Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. In: *Int. J. on Software Tools for Technology Transfer (STTT)* Sp. Sec. CPN 04/05. URL: http://dx.doi.org/10.1007/s10009-007-0038-x (cited on page 315).

Jokikyyny, T. and C. Lassenius (1999). Using the Internet to Communicate Software Metrics in a Large Organization. In: *Proc. Globecom'99* (cited on page 335).

Joshi-Tope, G. et al. (2005). Reactome: a knowledgebase of biological pathways. In: *Nucleic Acids Research* 33.suppl_1, pages D428–432. URL: 10.1093/nar/gki072 (cited on page 312).

Jossic, A., M. Del Fabro, J.-P. Lerat, J. Bezivin and F. Jouault (2007). Model Integration with Model Weaving: a Case Study in System Architecture. In: pages 79–84. DOI: 10.1109/ICSEM.2007.373336 (cited on page 173).

Jouault, F. and I. Kurtev (2006). On the architectural alignment of ATL and QVT. In: *Proceedings of the 2006 ACM symposium on Applied computing*. SAC '06. Dijon, France: ACM, pages 1188–1195. ISBN: 1-59593-108-2. DOI: http://doi.acm.org/10.1145/1141277.1141561. URL: http://doi.acm.org/10.1145/1141277.1141561 (cited on page 41).

Kanehisa, M., S. Goto, M. Hattori, K. Aoki-Kinoshita, M. Itoh, S. Kawashima, T. Katayama, M. Araki and M. Hirakawa (2006). From genomics to chemical genomics: new developments in KEGG. In: *Nucleic Acids Research* 34.suppl_1. URL: 10.1093/nar/gkj102 (cited on page 312).

Kang, K., S. Cohen, J. Hess, W. Nowak and S. Peterson (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (cited on pages 4, 50 sq.).

Kang, K., J. Lee and P. Donohoe (July 2002). Feature-oriented product line engineering. In: *Software, IEEE* 19.4, pages 58 –65. ISSN: 0740-7459. DOI: 10.1109/MS.2002.1020288 (cited on page 51).

Kang, K. C. (2009). Keynote Address: FODA: Twenty Years of Perspective on Feature Models. In: *Software Product Line Conference (SPLC) 2009*. San Francisco, CA, USA. URL: http://www.sei.cmu.edu/splc2009/files/SPLC_FODA_20090828-final.pdf (cited on pages 50 sq.).

Kang, K. C., S. Kim, J. Lee, K. Kim, E. Shin and M. Huh (1998). FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. In: *Ann. Software Eng.* 5, pages 143–168 (cited on page 51).

Karp, P., C. Ouzounis, C. Moore-Kochlacs, L. Goldovsky, P. Kaipa, D. Ahren, S. Tsoka, N. Darzentas, V. Kunin and N. Lopez-Bigas (2005). Expansion of the BioCyc collection of pathway/genome databases to 160 genomes. In: *Nucleic Acids Research* 33.19, pages 6083–6089. URL: 10.1093/nar/gki892 (cited on page 312).

Kelly, S., K. Lyytinen and M. Rossi (1996). MetaEdit+ A fully configurable multi-user and multi-tool CASE and CAME environment. In: *Advanced Information Systems Engineering*. Edited by P. Constantopoulos, J. Mylopoulos and Y. Vassiliou. Volume 1080. Lecture Notes in Computer Science. 10.1007/3-540-61292-0_1. Springer Berlin / Heidelberg, pages 1–21. ISBN: 978-3-540-61292-6. URL: http://dx.doi.org/10.1007/3-540-61292-0_1 (cited on page 364).

Kempkens, R., P. Rösch, L. Scott and J. Zettel (2000). Instrumenting Measurement Programs with Tools. In: *PROFES*. Edited by F. Bo-

marius and M. Oivo. Volume 1840. Lecture Notes in Computer Science. Springer, pages 353–375. ISBN: 3-540-67688-0 (cited on page 335).

Kent, S. (2002). Model Driven Engineering. In: *Integrated Formal Methods, Third International Conference, IFM 2002, Turku, Finland, May 15-18, 2002, Proceedings*. Edited by M. J. Butler, L. Petre and K. Sere. Volume 2335. Lecture Notes in Computer Science. Springer, pages 286–298. ISBN: 3-540-43703-7 (cited on page 14).

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin (1997). Aspect-oriented programming. In: *ECOOP'97 - Object-Oriented Programming*. Edited by M. Akşit and S. Matsuoka. Volume 1241. Lecture Notes in Computer Science. Berlin/Heidelberg: Springer-Verlag. Chapter 10, pages 220–242. ISBN: 3-540-63089-9. DOI: 10.1007/BFb0053381. URL: http://dx.doi.org/10.1007/BFb0053381 (cited on page 70).

Kleppe, A. G., J. Warmer and W. Bast (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 032119442X (cited on page 16).

KMF Project (2011). *Kent Modeling Framework*. URL: http://www.cs.kent.ac.uk/projects/kmf/ (cited on page 167).

Komatsoulis, G., D. Warzel, F. Hartel, K. Shanbhag, R. Chilukuri, G. Fragoso, S. Coronado, D. Reeves, J. Hadfield, C. Ludet, et al. (2007). caCORE version 3: Implementation of a model driven, service-oriented architecture for semantic interoperability. In: *J Biomed Inform* (cited on page 330).

Komi-Sirviö, S., P. Parviainen and J. Ronkainen (2001). Measurement Automation: Methodological Background and Practical Solutions-A Multiple Case Study. In: *IEEE METRICS*. IEEE Computer Society, pages 306–316. ISBN: 0-7695-1043-4 (cited on page 335).

Kruchten, P. (1995). Architecture blueprints – The "4+1" view model of software architecture. In: *Tutorial proceedings on TRI-Ada '91: Ada's role in global markets: solutions for a changing complex world*. Anaheim, California, United States: ACM, pages 540–555. ISBN: 0-89791-705-7. DOI: 10.1145/216591.216611. URL: http:

//dl.acm.org/citation.cfm?id=216591.216611 (cited on pages 67 sq.).

Krull, M. et al. (2006). TRANSPATH(R): an information resource for storing and visualizing signaling pathways and their pathological aberrations. In: *Nucleic Acids Research* 34.suppl_1, pages D546–551. URL: 10.1093/nar/gkj107 (cited on pages 312, 315).

Kurtev, I., J. Bezivin, and M. Aksit (2002). Technological spaces: An initial appraisal. In: *In Tenth International Conference on Cooperative Information Systems (CoopIS), Federated Conferences Industrial Track, California.* (Cited on page 221).

Laguna, M. A., B. González-Baixauli and J. M. Marqués Corral (2008). *Feature Patterns and Multi-Paradigm Variability Models.* Technical report 2008/01. Departamento de Informática: Grupo GIRO. URL: http://www.giro.infor.uva.es/Publications/2008/LGM08 (cited on page 372).

Lavazza, L. and A. Agostini (2005). Automated Measurement of UML Models: an open toolset approach. In: *Journal of Object Technology* 4.4, pages 115–134 (cited on page 335).

Li, H., J. Gennari, J. Brinkley, et al. (2006). Model Driven Laboratory Information Management Systems. In: *AMIA Annu Symp Proc* 484, page 8 (cited on page 330).

Limón Cordero, R. N. (2010). Las vistas arquitectónicas de software y sus correspondencias mediante la gestión de modelos. SPA. PhD thesis. Universidad Politécnica de Valencia (cited on pages 68 sqq., 125, 146, 154, 245, 378).

Marca, D. A. and C. L. McGowan (1987). *SADT: structured analysis and design technique.* New York, NY, USA: McGraw-Hill, Inc. ISBN: 0-07-040235-3 (cited on page 13).

McAffer, J., P. VanderLei and S. Archer (2009). *OSGi and Equinox: Creating Highly Modular Java Systems.* Eclipse series. Addison-Wesley. ISBN: 9780321585714. URL: http://books.google.com/books?id=RjX8PQAACAAJ (cited on page 123).

McIlroy, M. D. (1968). Mass-produced software components. In: *Proc. NATO Conf. on Software Engineering, Garmisch, Germany* (cited on page 3).

Mellor, S. J., S. Kendall, A. Uhl and D. Weise (2004). *MDA Distilled*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc. ISBN: 0201788918 (cited on page 18).

Metacase, Corp. (2007). MetaEdit web site. In: URL: http://www.metacase.com/ (cited on page 364).

Microsoft (2003). *Visio 2003 SDK Documentation*. URL: http://msdn.microsoft.com/en-us/library/aa173161/office.11.aspx (cited on page 353).

Montero, F. and E. Navarro (2009). ATRIUM: Software Architecture Driven by Requirements. In: *Engineering of Complex Computer Systems, IEEE International Conference on* 0, pages 230–239. DOI: http://doi.ieeecomputersociety.org/10.1109/ICECCS.2009.21 (cited on page 350).

Mora, B., F. García, F. Ruiz, M. Piattini, A. Boronat, A. Gómez, J. Carsi and I. Ramos (2010). Software Generic Measurement Framework Based on MDA. In: *Latin America Transactions, IEEE (Revista IEEE America Latina)* 8.5, pages 605 –613. ISSN: 1548-0992. DOI: 10.1109/TLA.2010.5623515 (cited on page 390).

Mora, B., F. Garcia, F. Ruiz, M. Piattini, A. Boronat, A. Gomez, J. Carsi and I. Ramos (2011). Software Generic Measurement Framework Based on MDA. In: *Latin America Transactions, IEEE (Revista IEEE America Latina)* 9.1, pages 864 –871. ISSN: 1548-0992. DOI: 10.1109/TLA.2011.5876432 (cited on page 391).

Mora, B., F. Garcia, F. Ruiz, M. Piattini, A. Boronat, A. Gómez, J. Á. Carsí and I. Ramos (2007). Marco de Trabajo basado en MDA para la Medición Genérica del Software. In: *XII Jornadas de Ingeniería del Software y Bases de Datos. JISBD'07. Zaragoza. Spain.* Edited by X. Franch. URL: http://www.sistedes.es/sistedes/pdf/2007/JISBD-07-mora-mda.pdf (cited on page 390).

Mora, B., F. Garcia, F. Ruiz, M. Piattini, A. Boronat, A. Gómez, J. Á. Carsí and I. Ramos (2008a). JISBD2007-08: Software generic measurement framework based on MDA. In: *Latin America Transactions, IEEE (Revista IEEE America Latina)* 6.4, pages 363–370. ISSN: 1548-0992. DOI: 10.1109/TLA.2008.4815290 (cited on page 390).

Mora, B., F. García, F. Ruiz, M. Piattini, A. Boronat, A. Gómez, J. A. Carsí and I. Ramos (2008b). Software Measurement by Using QVT Transformations in an MDA Context. In: *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume DISI, Barcelona, Spain, June 12-16, 2008*. Edited by J. Cordeiro and J. Filipe, pages 117–124 (cited on page 390).

Motta, S. and V. Brusic (2004). Mathematical Modelling of the Immune System. In: edited by G. Ciobanu and G. Rozenberg. Modelling in Molecular Biology. Springer-Verlag Berlin, pages 193–218 (cited on page 313).

Navarro, E. (2007). ATRIUM: Architecture traced from requirements by applying a unified methodology. PhD thesis. Universidad de Castilla–La Mancha (cited on pages 307, 350 sq., 357 sq., 361).

Navarro, E. (2011). *MORPHEUS*. URL: http://www.dsi.uclm.es/personal/ElenaNavarro/research_atrium.htm (cited on page 350).

Navarro, E. and C. E. Cuesta (2008). Automating the Trace of Architectural Design Decisions and Rationales Using a MDD Approach. In: *Proceedings of the 2nd European conference on Software Architecture*. ECSA '08. Paphos, Cyprus: Springer-Verlag, pages 114–130. ISBN: 978-3-540-88029-5. DOI: http://dx.doi.org/10.1007/978-3-540-88030-1_10. URL: http://dx.doi.org/10.1007/978-3-540-88030-1_10 (cited on page 352).

Navarro, E., J. A. Mocholi, P. Letelier and I. Ramos (2006). A metamodeling approach for requirements specification. In: 47.5, pages 67–77 (cited on pages 350, 354).

Navarro, E., P. Letelier, D. Reolid and I. Ramos (2007a). Configurable Satisfiability Propagation for Goal Models Using Dynamic Compilation Techniques. In: *Advances in Information Systems Development*. Edited by W. Wojtkowski, W. G. Wojtkowski, J. Zupancic, G. Magyar and G. Knapp. 10.1007/978-0-387-70802-7_14. Springer US, pages 167–179. ISBN: 978-0-387-70802-7. URL: http://dx.doi.org/10.1007/978-0-387-70802-7_14 (cited on page 357).

Navarro, E., P. Letelier and I. Ramos (2007b). Requirements and Scenarios: Running Aspect-Oriented Software Architectures. In: *Software Architecture, Working IEEE/IFIP Conference on* 0, page 23. DOI: http://doi.ieeecomputersociety.org/10.1109/WICSA.2007.36 (cited on page 351).

Navarro Martínez, E., A. Gómez Llana, P. Letelier Torres and I. Ramos Salavert (2009). MORPHEUS: a supporting tool for MDD. In: *18th International Conference on Information Systems Development (ISD2009). Nanchang, China. September 16-19, 2009*. eprint: http://issi.dsic.upv.es/publications/archives/f-1249040391367/MORPHEUS_ISD_camera_ready.pdf (cited on page 391).

OMG (2002). *Request for Proposal: MOF 2.0 Query / Views / Transformations RFP*. Edited by Object Management Group. Object Management Group. URL: http://www.omg.org/cgi-bin/doc?ad/2002-04-10 (cited on page 41).

OMG (2003). *MDA Guide Version 1.0.1*. Object Management Group. URL: http://www.omg.org/docs/omg/03-06-01.pdf (cited on pages 15, 17, 310).

OMG (2004). *Common Object Request Broker Architecture: Core Specification*. Object Management Group. URL: http://www.omg.org/spec/CORBA/3.0.3/ (cited on page 15).

OMG (2005a). *Information Management Metamodel (IMM) RFP*. Object Management Group. URL: http://www.omg.org/cgi-bin/doc?ab/2005-12-2 (cited on page 40).

OMG (2005b). *Reusable Asset Specification Version 2.2*. Object Management Group. URL: http://www.omg.org/spec/RAS/2.2/ (cited on page 220).

OMG (2006). *Meta Object Facility (MOF) 2.0 Core Specification*. Object Management Group. URL: http://www.omg.org/spec/MOF/2.0/ (cited on pages 34, 40, 356).

OMG (2008a). *MOF 2.0 QVT final adopted specification (ptc/05-11-01)*. Edited by Object Management Group. Object Management Group. URL: http://www.omg.org/spec/QVT/1.0/ (cited on pages 20, 22, 91, 352).

OMG (2008b). *Semantics of Business Vocabulary and Business Rules (SBVR) Version 1.0*. Object Management Group. URL: http://www.omg.org/spec/SBVR/1.0/ (cited on page 40).

OMG (2008c). *Software & Systems Process Engineering Meta-Model Specification (SPEM) Version 2.0*. Object Management Group. URL: http://www.omg.org/spec/SPEM/2.0/ (cited on pages 84, 220, 350).

OMG (2009). *Request for Proposal: Common Variability Language (CVL) RFP*. Edited by Object Management Group. Object Management Group. URL: http://www.omg.org/cgi-bin/doc?ad/2009-12-03 (cited on page 371).

OMG (2010a). *OCL 2.2 Specification*. Object Management Group. URL: http://www.omg.org/spec/OCL/2.2/ (cited on page 20).

OMG (2010b). *UML 2.0 infrastructure specification (formal/05-07-05)*. Object Management Group. URL: http://www.omg.org/spec/UML/2.3/ (cited on pages 20, 227).

OMG (2011a). *Business Process Model and Notation (BPMN) Version 2.0*. Object Management Group. URL: http://www.omg.org/spec/BPMN/2.0/ (cited on page 40).

OMG (2011b). *Object Management Group*. URL: http://www.omg.org (cited on page 15).

OMG (2011c). *OMG MOF 2 XMI Mapping Specification Version 2.4.1*. Edited by Object Management Group. Object Management Group. URL: http://www.omg.org/spec/XMI/2.4.1/ (cited on page 37).

OSGi Alliance (2008). *OSGi Service Platform Core Specification*. OSGi Alliance. URL: http://www.osgi.org/Specifications/HomePage/ (cited on page 118).

OSLO, Open Source Library for OCL Project (2011). *OSLO website*. URL: http://oslo-project.berlios.de/ (cited on page 43).

Palza, E., C. Fuhrman and A. Abran (2003). Establishing a Generic and Multidimensional Measurement Repository in CMMI context. In: *Software Engineering Workshop, Annual IEEE/NASA Goddard* 0, page 12. DOI: http://doi.ieeecomputersociety.org/10.1109/SEW.2003.1270721 (cited on page 335).

Perez, J., E. Navarro, P. Letelier and I. Ramos (2006). A Modelling Proposal for Aspect-Oriented Software Architectures. In: *Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*. Washington, DC, USA: IEEE Computer Society, pages 32–41. ISBN: 0-7695-2546-6. DOI: 10.1109/ECBS.2006.12. URL: http://dl.acm.org/citation.cfm?id=1126179.1126186 (cited on pages 361 sq.).

Pratt & Whitney (2011). *Pratt & Whitney website*. URL: http://www.pw.utc.com/ (cited on page 276).

Pressman, R. S. (2001). *Software Engineering: A Practitioner's Approach*. 5th edition. McGraw-Hill Higher Education. ISBN: 0072496681 (cited on pages 249 sqq.).

Pérez, J., N. Ali, J. Carsí and I. Ramos (2006). Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In: *Component-Based Software Engineering*. Edited by I. Gorton, G. Heineman, I. Crnkovic, H. Schmidt, J. Stafford, C. Szyperski and K. Wallnau. Volume 4063. Lecture Notes in Computer Science. 10.1007/11783565_9. Springer Berlin / Heidelberg, pages 123–138. ISBN: 978-3-540-35628-8. URL: http://dx.doi.org/10.1007/11783565_9 (cited on pages 352, 362).

Pérez, J., N. Ali, J. A. Carsí, I. Ramos, B. Álvarez, P. Sanchez and J. A. Pastor (2008). Integrating aspects in software architectures: PRISMA applied to robotic tele-operated systems. In: *Information and Software Technology* 50.9-10, pages 969 –990. ISSN: 0950-5849. DOI: DOI:10.1016/j.infsof.2007.08.007. URL: http://www.sciencedirect.com/science/article/B6V0B-4PNM477-1/2/1d72dfd5c6aa472d17d68e907ce423ff (cited on pages 70, 238, 269).

Pérez Benedí, J. (2006). PRISMA: Aspect-Oriented Software Architectures. PhD thesis. Departamento de Sistemas Informáticos y Computación. Universidad Politécnica de Valencia. URL: http://hdl.handle.net/10251/1928 (cited on pages 70, 161, 220, 222, 269).

Rashid, A., J. Royer and A. Rummler (2011). *Aspect-Oriented, Model-Driven Software Product Lines: The AMPLE Way*. Cambridge Uni-

versity Press. ISBN: 9780521767224. URL: http://books.google.es/books?id=p5lgVeg7wsMC (cited on page 374).

Richters, M. and M. Gogolla (2000). Validating UML Models and OCL Constraints. In: *Proc. 3rd International Conference on the Unified Modeling Language (UML)*. Edited by A. Evans, S. Kent and B. Selic. Volume 1939. Springer-Verlag, pages 265–277. URL: citeseer.ist.psu.edu/richters00validating.html (cited on page 21).

Rolls-Royce Group plc (2011). *Rolls-Royce Group plc website*. URL: http://www.rolls-royce.com/ (cited on page 276).

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy and W. Lorenson (Oct. 1991). *Object-Oriented Modeling and Design*. Prentice Hall, Inc. ISBN: 0-13-629841-9 (cited on page 13).

Sardinha, A., R. Chitchyan, N. Weston, P. Greenwood and A. Rashid (2009). EA-Analyzer: Automating Conflict Detection in Aspect-Oriented Requirements. In: *Automated Software Engineering, 2009. ASE '09. 24th IEEE/ACM International Conference on*, pages 530–534. DOI: 10.1109/ASE.2009.31 (cited on page 374).

Scotto, M., A. Sillitti, G. Succi and T. Vernazza (2004). A relational approach to software metrics. In: *SAC*. Edited by H. Haddad, A. Omicini, R. L. Wainwright and L. M. Liebrock. ACM, pages 1536–1540. ISBN: 1-58113-812-1 (cited on page 335).

Selic, B. (2003). The Pragmatics of Model-Driven Development. In: *IEEE Software* 20.5, pages 19–25. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231146 (cited on pages 31, 349).

Shakil Khan, S., P. Greenwood, A. Garcia and A. Rashid (2008). On the Impact of Evolving Requirements-Architecture Dependencies: An Exploratory Study. In: *Advanced Information Systems Engineering*. Edited by Z. Bellahsène and M. Léonard. Volume 5074. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pages 243–257. ISBN: 978-3-540-69533-2. URL: http://dx.doi.org/10.1007/978-3-540-69534-9_19 (cited on page 374).

Shaw, M. (1984). Abstraction Techniques in Modern Programming Languages. In: *IEEE Software* 1, pages 10–26. ISSN: 0740-7459. DOI:

http://doi.ieeecomputersociety.org/10.1109/MS.1984.229453 (cited on page 67).

Shaw, M. and P. Clements (2006). The Golden Age of Software Architecture. In: *IEEE Softw.* 23 (2), pages 31–39. ISSN: 0740-7459. DOI: 10.1109/MS.2006.58. URL: http://portal.acm.org/citation.cfm?id=1128592.1128707 (cited on pages 146, 245).

She, S., R. Lotufo, T. Berger, A. Wasowski and K. Czarnecki (2010). The Variability Model of The Linux Kernel. In: *VaMoS*. Edited by D. Benavides, D. S. Batory and P. Grünbacher. Volume 37. ICB-Research Report. Universität Duisburg-Essen, pages 45–51. URL: http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf (cited on page 377).

Sommerville, I. (2004). *Software Engineering (International Computer Science Series)*. 7th edition. Addison Wesley. ISBN: 0321210263. URL: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/0321210263 (cited on page 4).

Song, Y., E. Kawas, B. Good, M. Wilkinson and S. Tebbutt (2007). DataBiNS: a BioMoby-based data-mining workflow for biological pathways and non-synonymous SNPs. In: *Bioinformatics* 23.6, page 780 (cited on page 330).

Steinberg, D., F. Budinsky, M. Paternostro and E. Merks (2009). *EMF: Eclipse Modeling Framework*. 2nd edition. Boston, MA: Addison-Wesley. ISBN: 978-0-321-33188-5 (cited on pages 34, 126).

Sterling, L. and E. Shapiro (1986). *The Art of Prolog*. Cambridge (MA): MIT Press (cited on page 373).

Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. 2nd edition. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201745720 (cited on page 161).

Taubner, C., B. Mathiak, A. Kupfer, N. Fleischer and S. Eckstein (2006). Modelling and Simulation of the TLR4 Pathway with Coloured Petri Nets. In: *2006. EMBS '06. 28th Annual International Conference of the IEEE*, pages 2009–2012. ISSN: 1557-170X (cited on pages 316 sq.).

Trujillo, S. (2007). Feature Oriented Model Driven Product Lines. PhD thesis. School of Computer Sciences, University of the Basque Country (cited on pages 273, 373).

Tsang, E. (1995). *Foundations of Constraint Satisfaction*. Academic Press (cited on page 274).

Vépa, E, J Bézivin, H Brunelière and Jouault (2006). Measuring model repositories. In: *In: Model Size Metrics Workshop at the MoDELS/UML* (cited on pages 335 sq.).

Weston, N., R. Chitchyan and A. Rashid (2009). A framework for constructing semantically composable feature models from natural language requirements. In: *Proceedings of the 13th International Software Product Line Conference*. SPLC '09. San Francisco, California: Carnegie Mellon University, pages 211–220. URL: http://dl.acm.org/citation.cfm?id=1753235.1753265 (cited on page 374).

Ziegler, M. (2007). Implementierung eines Transformationsmoduls in Java zur Abbildung von Signaltransduktions-Instanzen auf CPN-Instanzen. Master's thesis. Technische Universität Braunschweig, page 140 (cited on page 319).

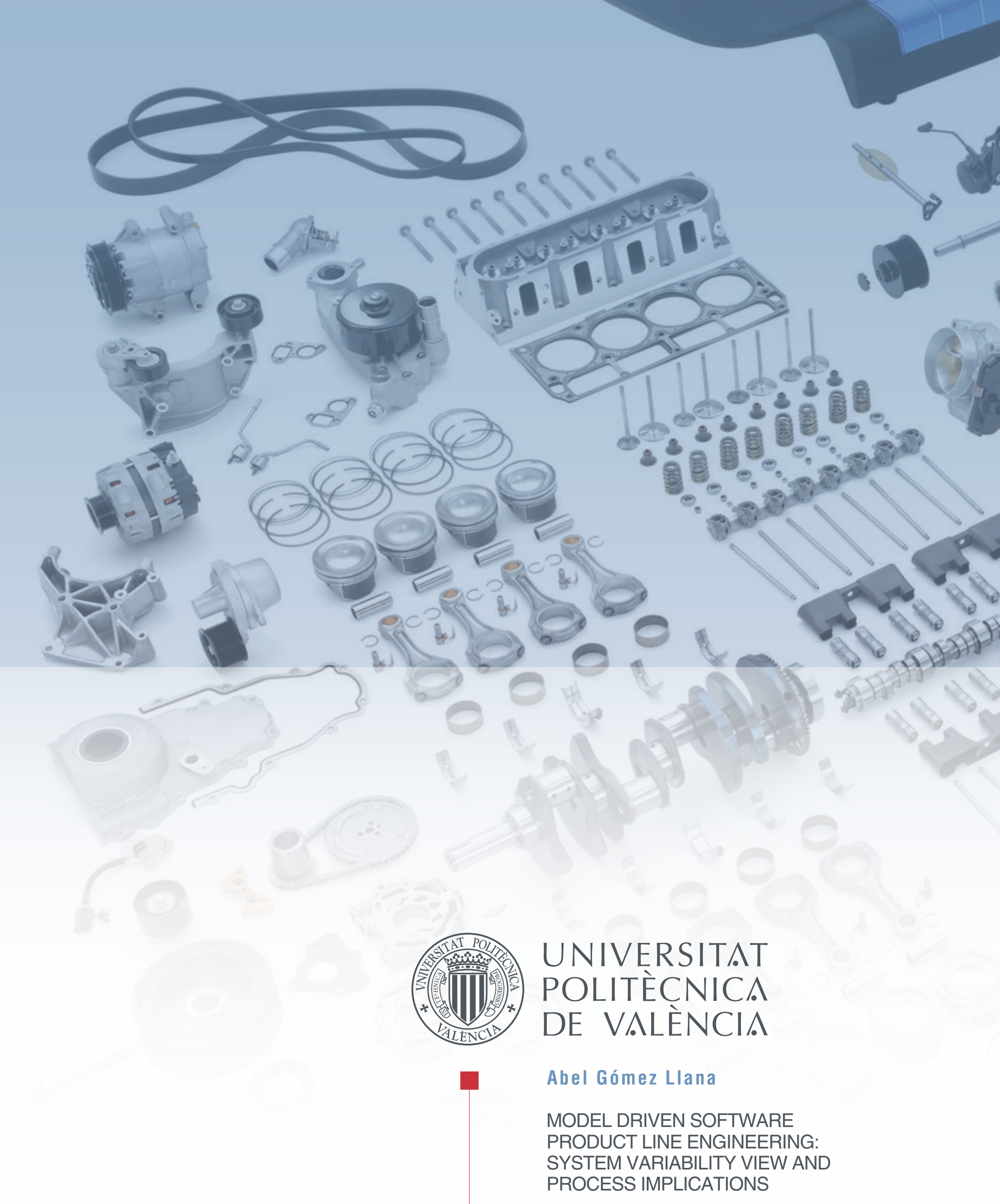
Zschaler, S., P. Sanchez, J. Santos, M. Alferez, A. Rashid, L. Fuentes, A. Moreira, J. Araujo and U. Kulesza (2010). VML* – A Family of Languages for Variability Management in Software Product Lines. In: *Engineering*, pages 82–102. URL: http://eprints.lancs.ac.uk/42521/ (cited on page 376).

SHORT VITA

ABEL GÓMEZ LLANA received his degree on Computer Science (*Ingeniero en Informática*) at the *Universidad Politécnica de Valencia* in 2005. In 2006, he stayed at the Institute for Information Systems (Institut für Informationssysteme) at Technische Universität Braunschweig (Germany), where he began his research works in bioinformatics. He received a master's degree in Software Engineering, Formal Methods and Information Systems in 2008. Abel Gómez has hold several research, teaching, and collaboration grants since 2004; including the research and teaching FPU fellowship (*Fomación de Profesorado Universitario*) funded by the Spanish Ministry of Education and Science (*Ministerio de Educación y Ciencia*) between 2007–2011. Currently, he is a senior research technician at the *Departamento de Sistemas Informáticos y Computación*, *Universidad Politécnica de Valencia*. His areas of interest include model-driven software development, model transformations, and their applicability in diverse fields such as bioinformatics, software product lines, feature modeling and variable content documents.

*The good thing about science is that
it's true whether or not you believe in it*

— Neil deGrasse Tyson
American astrophysicist and science communicator, 1958–



UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Abel Gómez Llana

MODEL DRIVEN SOFTWARE PRODUCT LINE ENGINEERING: SYSTEM VARIABILITY VIEW AND PROCESS IMPLICATIONS

PhD thesis | March 2012 | Supervised by Dr. Isidro Ramos Salavert | Programa de doctorado en programación declarativa e ingeniería de la programación