Cristóbal Costa-Soria

DYNAMIC EVOLUTION AND RECONFIGURATION OF SOFTWARE ARCHITECTURES THROUGH ASPECTS

PhD Thesis. May 2011

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

SUPERVISORS:
Dr. Jose Ángel Carsí Cubel
Dr. Jennifer Pérez Benedí

DYNAMIC EVOLUTION AND RECONFIGURATION OF SOFTWARE ARCHITECTURES THROUGH ASPECTS

Cristóbal Costa-Soria

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DEPARTMENT OF INFORMATION SYSTEMS AND COMPUTATION

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

Supervised by
Dr. Jennifer Pérez Benedí
Dr. Jose A. Carsí Cubel

May 2011

Supervisors

Dr. Jennifer Pérez Benedí
E.U. Informática, Technical University of Madrid (Spain)

Dr. Jose Ángel Carsí Cubel
Dept. Information Systems & Computation, Univ. Politécnica de Valencia (Spain)

This thesis has been partially supported by the Spanish Ministry of Education and Science through National Research, Development and Innovation projects DYNAMICA (TIC2003-07804-C05-01), META/Moment (TIN2006-15175-C05-01), by the European Regional Development Fund (FEDER), by the Comunidad de Madrid and the Universidad Rey Juan Carlos under the IASOMM project (URJC-CM-2007-CET-1555), and by the Conselleria d'Educació i Ciència (Generalitat Valenciana) under the contract BFPI06/227.

Credits of the pictures
Álvaro Caminero, toxicpicnic.com (cover, pp. 21 and 131)
Cristóbal Costa-Soria (pp. 59, 351, 377)

ABSTRACT

Change is an intrinsic property of software. A software system, during its lifetime, may require several updates, improvements, or new features. If these change requirements are not addressed, the risk of becoming a useless system increases. In fact, this is a challenging issue of safety- and mission-critical software systems, which cannot be stopped to perform maintenance or evolution operations due to their continuous operation. To reduce the aging of these critical systems, they must be provided with mechanisms enabling their dynamic evolution, i.e. the support of changes on their structure and behaviour while they remain in operation.

This thesis is concerned with the design of a framework to build architecture-based, dynamically evolvable, software systems. The fact that this framework is a software architecture based approach provides the following advantages: (i) it offers a high-level of abstraction for describing dynamic changes; (ii) it allows varying the level of system description; and (iii) it advantages from the existing support for system modelling, code-generation, and formal analysis provided by architecture description languages.

The framework presented in this thesis, called *Dynamic PRISMA*, is characterized by the combination of two levels of dynamism: Dynamic Reconfiguration, which addresses changes at the configuration level (i.e. the architectural configuration), and Dynamic Type Evolution, which addresses changes at the type-level (i.e. the specification of architectural types and instances). This combination is one of the major contributions of this thesis: thus a system is not only able to reconfigure at runtime the building blocks it is composed of (i.e. architectural types), but also to redefine these building blocks (or introduce new ones) at runtime.

Another contribution of the thesis is the identification of the concerns related to dynamic evolution and their integration in the framework through aspects. This improves the separation of concerns and allows us to change reconfiguration specifications, evolution mechanisms, or the business logic independently of each other.

A third contribution of this thesis is how this dynamism is supported: reconfiguration through autonomic capabilities, which provides proactivity according to either internal or external stimuli; and type evolution through asynchronous reflection, which enables the modification of a type specification and the transformation of their instances at different rates (i.e. when they are ready for evolution). Specifically, the asynchronous evolution semantics is precisely described by means of graph transformations. This formalism has been chosen because it naturally models both the system architecture and its asynchronous evolution.

The work presented in this thesis is illustrated through a case study from the robotics domain; an area which could potentially benefit from the results of this thesis.

KEYWORDS

software architectures, dynamic evolution, dynamic reconfiguration, dynamic type evolution, dynamic instance evolution, asynchronous evolution, runtime evolution, self-managed systems, autonomic computing, computational reflection, aspect-oriented software development

RESUMEN

El cambio es una propiedad intrínseca del software. Un sistema software, a lo largo de su vida útil, puede necesitar actualizaciones, mejoras o la integración de nuevas características. Si estas necesidades de cambio no son cubiertas, el riesgo de que el sistema software deje de ser útil aumenta. Esto supone un reto para los sistemas críticos, los cuales no pueden ser detenidos para realizar operaciones de mantenimiento o evolución debido a que deben estar continuamente operativos. Para reducir el envejecimiento de dichos sistemas, éstos deben incorporar mecanismos que les permitan evolucionar dinámicamente, i.e. tolerar cambios tanto estructurales como de comportamiento mientras están operativos.

Esta tesis aborda el diseño de una infraestructura para la construcción de sistemas software dinámicamente evolucionables y basados en arquitecturas software. Las razones que han motivado el uso de un enfoque basado en arquitecturas software son: (i) proporcionan un alto nivel de abstracción para definir cambios dinámicos; (ii) permiten variar el nivel de descripción del sistema; y (iii) permiten reutilizar las herramientas existentes para modelado de sistemas, generación automática de código, y análisis formal proporcionadas por los lenguajes de descripción de arquitecturas.

El marco presentado en esta tesis, llamado *Dynamic PRISMA*, se caracteriza por la combinación de dos niveles de dinamismo: Reconfiguración Dinámica, que aborda los cambios a nivel de configuración (i.e. la configuración arquitectónica), y Evolución Dinámica de Tipos, que aborda los cambios a nivel de tipos (i.e. la especificación de tipos arquitectónicos e instancias). Esta combinación es una de las mayores contribuciones de esta tesis: así, un sistema no es solamente capaz de reconfigurar durante su ejecución los elementos constructivos que lo forman (i.e. los tipos arquitectónicos), sino también de redefinir dichos elementos constructivos (o introducir otros) durante su ejecución.

Otra contribución de la tesis es la identificación de las funcionalidades relacionadas con la evolución dinámica y su integración a través de aspectos. Esto mejora la separación de funcionalidades y permite cambiar de forma independiente entre sí las especificaciones de reconfiguración, los mecanismos de evolución, o la lógica de negocio.

Una tercera contribución es cómo este dinamismo se ha soportado: la reconfiguración a través de capacidades autonómicas, aportando así proactividad en función de estímulos internos y/o externos; y la evolución de tipos a través de la reflexión asíncrona, permitiendo así modificar la especificación de un tipo y la transformación de sus instancias en distintos tiempos (i.e. cuando éstas están listas para su evolución). Además, la semántica de la evolución asíncrona se ha formalizado a través de transformaciones de grafos, lo que ha permitido modelar de forma natural tanto la arquitectura de un sistema como su evolución asíncrona.

Por último, el trabajo presentado en esta tesis se ha ilustrado a través de un caso de estudio del dominio robótico; un área que podría verse potencialmente beneficiada con los resultados de esta tesis.

PALABRAS CLAVE

arquitecturas software, evolución dinámica, reconfiguración dinámica, evolución dinámica de tipos, evolución de instancias, evolución asíncrona, evolución en tiempo de ejecución, sistemas auto-gestionados, computación autonómica, reflexión computacional, desarrollo de software orientado a aspectos

RESUM

El canvi és una propietat intrínseca del programari. Un sistema informàtic, al llarg de la seua vida útil, pot necessitar actualitzacions, millores, o la integració de noves característiques. Si aquestes necessitats de canvi no són cobertes, el risc de que un sistema informàtic deixe d'ésser útil augmenta. Açò esdevé un repte per als sistemes crítics, els quals no poden ser parats per a realitzar operacions de manteniment o evolució a causa de que han d'estar contínuament operatius. Per a reduir l'envelliment d'aquests sistemes, s'han d'incorporar mecanismes que els permeten evolucionar dinàmicament, açò és, tolerar canvis tant estructurals com de comportament mentre estan operatius.

Aquesta tesis aborda el disseny d'una infraestructura per a la construcció de sistemes software dinàmicament evolucionables i basats en arquitectures de programari. Les raons que han motivat la utilització d'una aproximació basada en arquitectures de programari són les següents: (i) proporcionen un alt nivell d'abstracció per a definir canvis dinàmics; (ii) permeten variar el nivell d'abstracció dels sistemes complexes; y (iii) permeten reutilitzar el suport existent per al modelatge de sistemes, la generació automàtica de codi, i l'anàlisi formal proporcionat pels llenguatges de descripció d'arquitectures.

La infraestructura presentada a aquesta tesi, denominada *Dynamic PRISMA*, es caracteritza per la combinació de dos nivells de dinamisme: Reconfiguració Dinàmica, que tracta els canvis a nivell de configuració (i.e. la configuració arquitectònica), i l'Evolució Dinàmica de Tipus, que tracta els canvis a nivell de tipus (i.e. la especificació de tipus arquitectònics e instàncies). Aquesta combinació és una de les majors contribucions d'aquesta tesi: així, un sistema no és solament capaç de reconfigurar durant la seua execució els elements constructius amb que està format (i.e. els tipus arquitectònics), sinó també de redefinir aquests elements constructius (o introduir-ne d'altres) durant la seua execució.

Una altra contribució de la tesi és la identificació de les funcionalitats relacionades amb l'evolució dinàmica i la seua integració a través d'aspectes. Açò millora la separació de funcionalitats i permet canviar de forma independent les especificacions de reconfiguració, els mecanismes d'evolució, o la lògica de negoci.

Una tercera contribució és cóm aquest dinamisme s'ha suportat: la reconfiguració a través de capacitats autonòmiques, aportant així proactivitat en funció d'estímuls interns i/o externs; i la evolució de tipus mitjançant la reflexió asíncrona, permetent així modificar l'especificació d'un tipus i la transformació de les seues instàncies en distints temps (i.e. quan aquestes estiguen preparades per a evolucionar). A més a més, la semàntica de l'evolució asíncrona s'ha formalitzat mitjançant transformacions de grafs, el que ha permès modelar de forma natural tant l'arquitectura d'un sistema com la seua evolució asíncrona.

Per últim, el treball presentat a aquesta tesi s'ha il·lustrat mitjançant un cas d'estudi del domini robòtic; un àrea que podria veure's potencialment beneficiada amb els resultats d'aquesta tesi.

PARAULES CLAU

arquitectures de programari, evolució dinàmica, reconfiguració dinàmica, evolució dinàmica de tipus, evolució d'instàncies, evolució asíncrona, evolució en temps d'execució, sistemes auto-gestionats, computació autonòmica, reflexió computacional, desenvolupament de programari orientat a aspectes

Dedicatoria

A mi amada Sagrario, por acompañarme en este camino con tanta paciencia, cariño y apoyo incondicional; pero sobretodo, por hacer que cada momento sea tan especial.

A mi padre, mi suegro Pío Andrés y mis abuelos Vicente y Cristóbal, por todas sus sabias enseñanzas y los felices momentos que compartí con ellos, valiosos bienes que me han acompañado todo este tiempo.

A mi madre, mi hermano Raúl y Sonia, por haberme ayudado a olvidar los malos momentos, por sus ánimos, y por lograr arrancar mi sonrisa.

A mis tíos Pepe, Mª Isabel y Vicente, por despertar en mí la pasión por la Ciencia desde mi más temprana infancia y que aún pervive en mí.

A toda mi familia: mi abuela Rosa, mis tíos Antonio y Susi, mi suegra Sagrario, y todos mis cuñados y cuñadas, por su comprensión y apoyo durante todos estos meses que han tenido que prescindir de mí.

PREFACE

This thesis gathers together the most important results of more than five years of intensive research work. Simplicity, one of the fundamental principles of Science, has been always present in the developments of this work. At the beginnings of a research work, simplicity is not easy to achieve: the problem to address is generally blurred, complex and with a wide range of directions to explore. This exploration, which is inherent to any research, should be conducted to solve the research problem in a simple and clean way. However, the way to reach a solution is not unique and several ones may still exist waiting to be discovered.

This thesis describes the path that has been followed to deal with the *dynamic evolution of architecture-based systems* and the advantages it presents to other alternative solutions. Many results may look obvious, yet they were not so easy to achieve at the beginning. These results may reflect that I have succeeded in finding a simple, clean way to a facet of the research problem. However, other results may be difficult to understand, which may reflect that I have not succeeded in finding a simpler way. I am not completely happy with a lot of things (as I am assured it is the case for most writers), but time has come to finish the work. I hope you will enjoy it and find it interesting; or, that at least, it may raise questions that may contribute to advance in the area.

Manises, 20th February 2011

Acknowledgements

I would like to acknowledge the people who have helped me during these years, and who have shared with me their time, knowledge and friendship.

I want to express my sincere gratitude and appreciation to my supervisors, *Jennifer Pérez* and *Jose Angel Carsí*. Particularly, I would like to thank *Jennifer* for her encouragement, her availability (also in weekends) in almost all deadlines, her deep observations, and specially, her good sense of humour when reviewing my papers. I would like to thank *Jose Angel* for his guidance and support, but more importantly, to give me enough confidence and freedom to conduct my research.

I would like to thank everyone within the *Software Engineering and Information Systems* research group, for providing a productive and friendly environment. Special thanks must go to *Isidro Ramos* for leading the group, his unconditional support to all of us, and for driving us towards the most challenging topics. Special thanks must go to *Jose Hilario Canós*, *Javier Jaén*, *Mª Carmen Penadés*, *Patricio Letelier*, *Emilio Insfrán* and *Silvia Abrahao*, for their support and encouragement in this thesis. Additional thanks to *Adrián*, *Sonia*, *Emanuel*, *Lorena*, *David* and *Javier*, for providing a pleasant environment.

I especially want to thank these colleagues with whom I have shared not only office and lunch breaks, but also amusing and unforgettable moments: *Alejandro Català*, *Abel Gómez*, *Jose A. Mocholí*, *Manuel Llavador*, *Javier Jaén* and *Elena Navarro*. I cannot omit here *Nelly Condori*, *Gonzalo Rojas* and *Isabel Díaz*, who have made lunch breaks very pleasant, extremely amusing and de-stressing.

Additional thanks must be given to *Nour Ali* and *Carlos Solís*, with whom I have benefited from interesting discussions. I also would like to thank *Rogelio Limón*, *MªEugenia Cabello* and *Felipe López* for their great friendship. I am grateful, too, to *Carlos Millán*, *David Hervás*, and *Sheila Aliaga*, for their enthusiastic collaboration in the implementation of *Dynamic PRISMA* in PRISMANET. I cannot forget to mention here *Rafael Cabedo*, *Jose Manuel Cercós*, and *Javier Guillén*, with whom I shared enjoyable moments during the development of PRISMANET earlier prototypes.

Special thanks to *Carlos E. Cuesta*, a very inspiring researcher and person with a broad knowledge on diverse topics, for sharing with me the enthusiasm and his valuable comments about the topic, and for giving me confidence to continue with the work.

I would like to thank to all the wonderful people of the *Department of Computer Science of the University of Leicester*, who made me feel as if I was at home during my research stay. I especially want to thank *Reiko Heckel* for his helpful advice, for giving me different points of view that I had not previously considered, and for spending with me his valuable time. I also want to thank *José Fiadeiro* for his hospitality and for his valuable comments about my work.

In addition, I would like to thank *Inanna Catalá* and *Carlos E. Seaton* from *Global Metanoia*, for the flexibility, understanding and support they gave me in the last months of the writing process.

Finally, I would like to thank those anonymous reviewers who pointed out flaws, possible improvements and suggested new directions in my work.

TABLE OF CONTENTS

PART I: INTRODUCTION 23

CHAPTER 1. INTRODUCTION 25

1.1 Motivation 26

1.1.1 Self-Management 26

1.1.2 Runtime Maintenance 27

1.2 Overall Aim and Objectives 28

1.3 Research Methodology 30

1.4 Research Hypothesis 32

1.5 Thesis Overview 33

CHAPTER 2. CONTEXT 35

2.1 Introduction 35

2.2 Software Architectures 36

2.2.1 Definition 37

2.2.2 Basic Concepts 39

2.2.2.1 Component 39

2.2.2.2 Connector 40

2.2.2.3 Configuration 41

2.2.2.4 System 42

2.2.2.5 Port 42

2.2.2.6 Connection 42

2.2.2.7 Compositional Relationship 43

2.2.2.8 Other concepts 43

2.3 Aspect-Oriented Software Development 43

2.3.1 Aspect-Oriented Programming 44

2.3.2 Basic Concepts 46

2.3.3 Aspects in the Software Life Cycle 47

2.4 PRISMA 47

2.4.1 Model and ADL selection 48

2.4.2 Aspects as First-Class Citizens 49

2.4.3 PRISMA Architectural Elements 51

2.4.4 Levels of abstraction 53
2.4.5 Model-Driven Development Support 55
2.4.5.1 PRISMA in MOF 56
2.4.5.2 The PRISMA MDD Process 57
2.5 Conclusions 59

PART II: STATE OF THE ART 61

CHAPTER 3. DYNAMIC SOFTWARE EVOLUTION 63
3.1 Introduction 63
3.2 Software Evolution 63
3.2.1 Software Maintenance vs Software Evolution 64
3.2.2 Evolution as part of the Development Process 65
3.3 Dynamic Software Evolution 66
3.3.1 Changing software artefacts 66
3.3.2 Introducing changes at runtime 67
3.3.3 Dynamic Evolution: Definitions 68
3.3.4 Kinds of Dynamic Evolution 70
3.3.4.1 Granularity of changes 71
3.3.4.2 Activeness of change 72
3.4 Main Issues of Dynamic Evolution 73
3.4.1 Safe Stopping of Running Systems 74
3.4.1.1 Quiescence 74
3.4.1.2 Tranquillity 76
3.4.1.3 Other approaches for Safe Stopping 78
3.4.2 Updating Stateful Artefacts 80
3.4.2.1 No State Transfer 81
3.4.2.2 Delegated State Transfer 81
3.4.2.3 Automated State Transfer 82
3.5 Other approaches for dynamic change management 84
3.5.1 Control Systems 84
3.5.2 Autonomic Computing 85
3.5.3 Computational Reflection 89
3.6 Dynamic Evolution in Software Architectures 92
3.6.1 Dynamic Reconfiguration 92
3.6.2 Dynamic Evolution of Architectural Types 94
3.6.3 Combining both kinds of dynamism 95
3.7 Conclusions 95

CHAPTER 4. RELATED WORKS 97
4.1 Introduction 97
4.2 Dynamic Evolution Approaches 98
4.2.1 Formal, Dynamic ADLs for Reconfiguration 98
4.2.1.1 Process Algebra Formalisms 98
4.2.1.2 Graph-Based Formalisms 99
4.2.1.3 Reflection-Based Formalisms 101
4.2.2 Systems for Dynamic Change Support 102
4.2.2.1 Procedural and Object-Based Techniques 102
4.2.2.2 Dynamic Weaving Techniques in AOP 104
4.2.2.3 Component-Based Dynamic Frameworks 106
4.2.3 Self-Managed Software Architectures 110
4.2.3.1 Top-down approaches: Self-Adaptive Systems 110
4.2.3.2 Bottom-up approaches: Decentralized Architecture-Based Systems 116
4.2.4 AOSD & Evolution Concerns 119
4.3 Comparison of the different approaches 123
4.3.1 Description of the attributes selected 123
4.3.2 Comparison tables 126
4.4 Conclusions 130

PART III: DYNAMIC PRISMA 133

CHAPTER 5. CASE STUDY: AGROBOT 135
5.1 Introduction: Agricultural Robotics 135
5.2 Robotic Software Architectures 137
5.3 Dynamic Evolution in Robotic Software Architectures 138
5.4 Case Study: Agrobot, An Autonomous Robot for Plague control.. 139
5.4.1 Main Architecture 140
5.4.2 Composite Components: VisionSystem 142
5.4.3 Simple Components: ImageProcCard 145
5.4.4 Aspects: ImageProcSwController 147
5.5 Dynamic Evolution Requirements of the Agrobot 149
5.5.1 Dynamic reconfiguration scenario:
VisionSystem fault-tolerance support 150
5.5.2 Dynamic type evolution scenario:
Changing the ImageProcSoftware component 154
5.6 Conclusions 157

CHAPTER 6. AUTONOMIC RECONFIGURATION 159
6.1 Introduction 159
6.2 Characteristics of the approach 160
6.3 Reconfiguration management model 163
6.3.1 A control loop for self-reconfiguration 164
6.3.2 Aspects versus modules 166
6.4 Description of the autonomic reconfiguration aspects 167
6.4.1 The Monitoring Aspect 168
6.4.1.1 Introspection Services 170
6.4.1.2 Runtime Status Information 174
6.4.1.3 Event Interception Services 175
6.4.2 The Reconfiguration Analysis Aspect 179
6.4.2.1 Structure of the Reconfiguration Analysis Aspect 179
6.4.2.2 Reconfiguration Triggers 183
6.4.2.3 Configuration Transactions 185
6.4.2.4 Adding Inference Mechanisms 190
6.4.3 The Reconfiguration Coordination Aspect 190
6.4.3.1 Domain-Specific Reconfiguration Services 191
6.4.3.2 Transactional Management of Reconfiguration Plans 194
6.4.3.3 Generic Reconfiguration Services 202
6.4.4 The Reconfiguration Effector Aspect 214
6.4.4.1 Services for Safe Stopping 215
6.4.4.2 Services for Reconfiguration 216
6.4.4.3 Services for Updating, Recovery and Mobility 217
6.5 The Evolver Component 222
6.5.1 Structure of the Evolver Component 222
6.5.2 Support for Reactive Reconfigurations 225
6.5.3 Weaving the Reconfiguration Aspects Together 229
6.5.4 Evolver Specification 233
6.5.4.1 Evolver Template: The User-Defined Part 233
6.5.4.2 Evolver Mechanisms: The Generated Functionality 236
6.5.4.3 Consistence of Generated Code 238
6.6 Example: autonomic reconfiguration in the VisionSystem architecture 240
6.7 Conclusions & further works 244
6.7.1 Conclusions 244
6.7.2 Further works 246
6.7.3 Results 247

CHAPTER 7. DYNAMIC EVOLUTION OF ARCHITECTURAL TYPES.........249
7.1 Introduction 249
7.2 Basis of the dynamic evolution of architectural types..........250
7.2.1 Definitions of Type, Instance and Architectural Type.......... 250
7.2.2 Definition of the Dynamic Evolution of Architectural Types 252
7.3 Reflective Asynchronous Evolution of Architectural Types..........253
7.3.1 System-level evolutions vs type-level evolutions.......... 254
7.3.2 A reflective model for evolvable types 256
7.3.2.1 Reflection: The Abstract Model.......... 256
7.3.2.2 Type Meta-Instances: The Concrete Model.......... 257
7.3.2.3 Reification of Types.......... 259
7.3.2.4 Evolution Process Overview.......... 261
7.3.3 Evolving instances through transformations.......... 263
7.3.4 An asynchronous model for types evolution.......... 266
7.3.4.1 Modelling evolutions over time 266
7.3.4.2 Additional characteristics.......... 271
7.3.5 Description of the evolution infrastructure 273
7.3.5.1 Type-level Evolution 274
7.3.5.2 Instance-level Evolution.......... 299
7.3.6 Summary of the evolution process 313
7.4 Conclusions & further works..........316
7.4.1 Conclusions.......... 316
7.4.2 Further works 317
7.4.3 Results 318

CHAPTER 8. DESCRIPTION OF THE EVOLUTION SEMANTICS 321
8.1 Introduction..........321
8.2 Challenges of asynchronous evolution..........321
8.3 Evolution semantics..........322
8.3.1 Specification of evolution processes.......... 323
8.3.2 Version management: Evolution Tags 324
8.3.3 Formalisation of the evolution operations.......... 327
8.3.3.1 Architecture-based Concrete Syntax 328
8.3.3.2 Graph-based Abstract Syntax.......... 342
8.3.4 Discussion 349
8.4 Conclusions & further works..........350
8.4.1 Conclusions.......... 351
8.4.2 Further works 351
8.4.3 Results 352

PART IV: CONCLUSIONS & FURTHER WORK 353

CHAPTER 9. CONCLUSIONS 355
9.1 Conclusions 355
9.1.1 Contributions 356
9.1.2 Evaluation of the approach 359
9.1.3 Evaluation of the research 365
9.2 Results of the PhD 367
9.3 Further research 373
9.3.1 Model-Driven Development support for evolvable systems 373
9.3.2 Proactive non-programmed evolutions 374
9.3.3 Definition of evolution constraints 374
9.3.4 Coordination of decentralized Evolvers 375
9.3.5 Formal analysis 375
9.3.6 Tool support for advanced graph transformations 377

PART V: APPENDIXES 379

APPENDIX A. PRISMA SPECIFICATIONS OF THE VISIONSYSTEM 381
A.1 Specification of the VisionSystem type 381
A.1.1 Interfaces 381
A.1.2 Data Domains 382
A.1.3 External Functions 382
A.1.4 Architecture and configurations 383
A.1.5 Components 387
A.1.6 Connectors 389
A.1.7 Aspects 390
A.2 Reconfiguration Elements 395
A.2.1 Interfaces 395
A.2.2 Evolver Component 397
A.2.3 Reconfiguration Analysis Aspect 402
A.2.4 Reconfiguration Coordination Aspect 406
A.2.5 Architecture Monitoring Aspect 416
A.2.6 Architecture Effector Aspect 417
A.3 Type Evolution Elements 418
A.3.1 Data structures 418
A.3.2 Interfaces 422
A.3.3 Type Description Aspect 424
A.3.4 Type Evolution Aspect 428
A.3.5 Evolution Monitoring Aspect 429
A.3.6 Builder Aspect 430
A.3.7 Instance Evolution Planning aspect 432

A.3.8 Instance Monitoring aspect 434
A.3.9 Instance Effector aspect 434

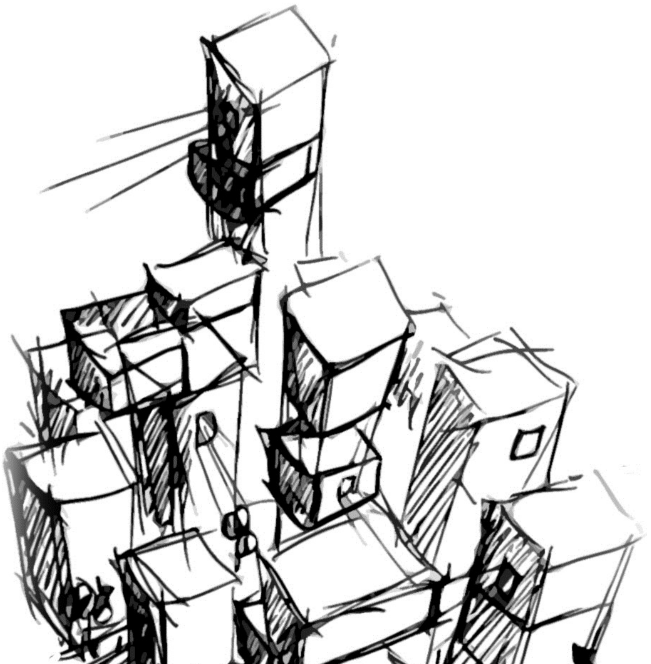
APPENDIX B. EXTENSIONS OF THE PRISMA AOADL 437
B.1 Lists 437
B.2 Iterations and Loops 438
B.3 Partial Definitions of Software Artefacts 439

LIST OF FIGURES 443

REFERENCES 447

PART I

INTRODUCTION



CHAPTER I

INTRODUCTION

The work presented in this thesis is concerned with the design of a framework to build architecture-based, dynamically evolvable, software systems. The goal is to support the evolution processes of such software systems that require updates, improvements or new features, but cannot be stopped to perform these operations due to their continuous operation. In addition, this support is also applicable to the building of self-managed software systems, which are able to autonomously change themselves while they remain operating. The framework presented in this thesis, called Dynamic PRISMA, is characterized by the combination of two levels of dynamism: Dynamic Reconfiguration, which addresses changes at the configuration level (i.e. the architectural configuration), and Dynamic Type Evolution, which addresses changes at the type-level (i.e. the specification of architectural types and instances). This combination is one of the major contributions of this thesis: thus a system is not only able to reconfigure at runtime the building blocks it is composed of (i.e. architectural types), but also to redefine these building blocks (or introduce new ones) at runtime. Another contribution of the framework is the way that it separates the concerns related to dynamic evolution and reconfiguration and encapsulates them into aspects, taking the advantages promoted by Aspect-Oriented Software Development. A third contribution of this framework is how it integrates the following capabilities: autonomic reconfigurations, reflective type descriptions, and asynchronous type evolutions.

The structure of this chapter is as follows: section 1.1 introduces the motivation of this work, section 1.2 explains the main goals of the thesis, section 1.3 presents the research methodology that has been followed during the development of the thesis, section 1.4 presents the research hypothesis, and section 1.5 summarizes the structure of the thesis.

1.1 Motivation

A well-known property of current and future software systems is their increasing size and complexity (SEI, 2006). Whereas technology evolves and provides more features, software systems increase their functionality and their non-functional requirements, such as distribution, decentralization, security, dependability, etc. This results in larger and complex software developments, which increase the costs of such systems.

One of the most promising techniques to deal with the design of large, complex software systems are Software Architectures (Perry & Wolf, 1992) (Taylor & Hoek, 2007). Software Architectures provide techniques for describing the structure of complex software systems (i.e. the key system elements and their organization). Their aim is to hide low-level details and help to understand the system. The structure of a software system is described in terms of architectural elements (components and connectors) and their interactions with each other. This structure can be formally described using an Architecture Description Language (ADL), which is used later to build the executable code of the software system. In addition, most ADLs generally support hierarchical composition (i.e. a composition hiding technique for defining systems of systems), which may be helpful for modelling large-scale complex systems in a scalable way. However, although Software Architectures help in the description and development of complex systems, this is not enough: the *management* and *maintenance* of these systems still requires a great effort (Perry, 2008) (Rombach, 2009).

1.1.1 Self-Management

To minimize the management effort of software systems, self-managed software architectures were proposed (Oreizy et al., 1999). According to the definition of Jeff Kramer and Jeff Magee:

> *A self-managed software architecture is one in which components automatically configure their interaction in a way that is compatible with an overall architectural specification and achieves the goals of the system.*
>
> *(Kramer & Magee, 2007)*

Examples of systems which require such degree of architectural dynamism are: (i) distributed systems with mobile elements, which frequently change the connections among its elements (Ali et al., 2006); (ii) multi-agent systems based on agreements (Santiago-Perez et al., 2009), which are open systems with a dynamic structure; or (iii) self-healing/fault-tolerant systems, which are provided with strategies to recover themselves from failures (Yurcik & Doss, 2001), (Dashofy et al., 2002).

However, the development of self-managed architectures still involves some challenges. One of them is to provide change management which reconfigures the software components, ensures application consistency and avoids undesirable transient behaviour (Kramer & Magee, 2007). This change management is also known as *Dynamic Reconfiguration* (Kramer & Magge, 1985), (Endler & Wei, 1992): the ability to change the architecture of a system while it is running. Although several works have been proposed (Bradbury et al., 2004), they generally do not scale well for large systems or do not explicitly consider the maintainability of the self-management system.

When a self-management infrastructure is designed, also its maintenance, scalability and flexibility must be taken into account. First, to improve maintenance, the concerns related to dynamic change should be isolated from functional concerns, as it has been stated from other works (Mens & Wermelinger, 2002), (Cazzola et al., 2007). Second, to increase scalability, self-management should be provided in a decentralized way, thus providing local autonomy. Finally, to be flexible enough, the self-management system should not only deal with goal-oriented proactive changes (i.e. driven autonomously), but also with reactive changes (i.e. driven externally), in order to cope with unanticipated situations.

For this reason, one of the objectives of this thesis is to propose a solution for supporting dynamic reconfiguration which overcomes these disadvantages.

1.1.2 Runtime Maintenance

Although self-managed approaches can reduce the management and configuration efforts of large software systems, this is not enough to deal with another property: the longevity expected for such systems.

Due to the high costs associated to the development of large software systems, they are expected to be usable for long periods of time. Then, unforeseen maintenance operations may be required by technology changes, new requirements or necessary corrective measures. This is due to the fact that large software systems are exposed to many sources of variability and they are integrated in changing environments. On the one hand, when more complex the system is, more probable is that unforeseen bugs appear during system execution, which must be removed. On the other hand, since it is impossible to predict all the features a system will require in the future, the system should be prepared to support the introduction of new features after its deployment. The maintenance of a software system is one of the most important phases of the software life cycle, since it will be present along its overall lifetime. From the beginning, complex software systems must be designed to facilitate their future maintenance: they must be flexible enough to support later modifications.

Nowadays, the current practice of introducing changes in running software systems is performed offline: the system is shutdown, and once the modification has been completed, the entire system is restarted to reflect the new changes. However, this solution has the disadvantage that the state and pending transactions of the running system are lost. In addition, some systems cannot stop their activity due to their critical nature. Examples of such systems are those that are in charge of critical infrastructures (e.g. related to military resources, energy, health or transports), those required to operate 365 days 24 hours (e.g. banking systems, manufacturing industry systems), or those that are not reachable (e.g. autonomous robots in space explorations).

In such cases, runtime maintenance support is needed. This is provided by *Dynamic Evolution*, a feature which supports the introduction of updates or new, unforeseen functionality on a running system. However, although self-managed systems allow the adaptation or reorganization of the system structure, they will not generally support the dynamic evolution of its types. What happens then if the behaviour of the components, or the reorganization algorithms, need to be changed without the system being stopped? In this regard, self-managed systems are incomplete.

For this reason, another objective of this thesis is to integrate the support for dynamic type evolution in self-managed approaches.

1.2 Overall Aim and Objectives

The objective of this work is to provide highly available systems (i.e. those systems that perform critical missions and cannot be stopped) with the ability to modify their structure (i.e. their software architecture) and/or their behaviour (i.e. their type definitions) at runtime.

This can be supported at different granularity levels. Granularity refers to the scale of the artefacts to be changed and can range from very coarse, through medium, to a very fine degree of granularity (Buckley et al., 2005). Coarse granularity refers to changes at the level of the system architecture; medium granularity refers to changes that impact the composition of modules/components, such as classes or objects; and fine granularity refers to changes on variables, methods or lines of code. Generally, existing approaches have only provided change management at a single granularity level. However, these levels complement each other and should be combined to provide the maximum degree of flexibility to critical software systems.

For these reasons, the overall aim of this thesis is to provide an integrated approach for supporting the three granularity levels of change in architecture-based systems. This can be achieved through the combination of *Dynamic Reconfiguration*, which addresses the reconfiguration of a software architecture at runtime (i.e. a coarse level), and *Dynamic Type Evolution*, which addresses the modification of types (i.e. specifications) and its types at runtime (i.e. at a medium and finer level). Since this approach should be integrated in a software development process, this work should not only cover the change management mechanisms, but also the integration of these mechanisms in the design of a system and their maintenance.

From a more general perspective, the overall aim of this thesis is formulated as follows:

> *To provide a framework to make easy the design, development, and maintenance of architecture-based software systems which are capable of changing their structure and behaviour at runtime without shutting them down.*

As a result, this framework should allow us to: (i) specify how the structure (i.e. the architecture) of a software system may change at runtime in response to different situations, like changes in the environment or changes to the business logic of the system; and (ii) describe how the architectural types of the software system can be evolved at runtime.

This goal is pursued by addressing the following specific objectives:

- To identify the main design strategies and mechanisms which enable the dynamic modification or recomposition of running software systems. These design strategies and mechanisms must be abstracted to a platform-independent perspective.
- To provide support for both degrees of architectural dynamism: dynamic reconfiguration of architectures and dynamic evolution of architectural types. The purpose is to provide the design and building of highly dynamic software systems with the highest degree of flexibility.
- To define a platform-independent model to specify the dynamic evolution/reconfiguration plans of evolvable systems.
- To preserve the *Separation of Concerns* principle (Parnas, 1972) as much as possible to facilitate the maintenance of the evolution concerns of highly dynamic systems.

- To automate the development process of evolvable systems. The system's architect should only deal with the specification of evolution/reconfiguration plans, and the supporting mechanisms should be provided by the infrastructure.

It is important to note that this work is based on the following assumption: the set of structural or behavioural changes that is going to be introduced in a system must be semantically compatible with the existing system. A software component can be dynamically replaced by another that performs a completely different purpose. However, the interacting components must be appropriately adapted, or replaced, to correctly interact with the new component. If these changes are not also provided by the architect or system developer, then the resulting system will not function properly, no matter of the evolution mechanisms used. This work is only concerned with the correct execution of dynamic changes, but not with trying to establish whether the changes proposed are correct.

1.3 Research Methodology

Research is an activity directed toward increased knowledge of either natural phenomena or the solution of an open issue, by following a scientific methodology. What scientific methodology to follow depends on which issues to investigate and on the field of science the research is carried out. In the area of Computer Science, there is a wide diversity of research methods that are being used (Wegner, 1976), (Dodig-Crnkovic, 2002), (Johnson, 2010). However, in the specific field of Software Engineering, there is no a well-established research method (Shaw, 2001), due to the diversity of research subjects (Glass et al., 2004) and the short history of Software Engineering (Xia, 1997).

The traditional *deductive methodology*, which is based on empiricism where hypotheses with clear falsification criteria can be identified, is not generally applicable in the Software Engineering field. The reason is that this research method is tailored to *analytic disciplines*, which are concerned with finding or discovering facts from natural phenomena. In contrast, the Software Engineering field is seen as a *synthetic discipline*, i.e. a discipline more oriented toward making and inventing new artefacts. Research in Software Engineering entails the creation of new models, methodologies and tools that are meant to help software developers, both to reduce development efforts and to be able to understand complex problems (Pressman, 2005).

The synthetic nature of software engineering aligns perfectly with the subject of study of the *design-science* paradigm (March & Smith, 1995), (Hevner et al., 2004): the scientific study of the artificial (Simon, 1996), as opposite to the study of the natural. Design science is essentially a problem-solving methodology that seeks to create and evaluate artefacts intended to solve identified problems. It focuses on the usefulness or utility of a method or artefact rather than on its truth, taking into account real-world constraints and practical considerations. Design science helps in managing the complexity linked to the design of useful artefacts in domain areas in which existing theory or previous knowledge is often insufficient. This is the key difference among design science and routine design: it addresses important unsolved problems in unique or innovative ways, or solved problems in more effective or efficient ways (Hevner et al., 2004).

As stated by Hevner et al., the fundamental principle behind *design-science research* is that *the knowledge and understanding of a design problem and its solution are acquired in the building and application of an artefact* (Hevner et al., 2004). According to this principle, Hevner et al. (Hevner et al., 2004) propose seven guidelines to help information systems researchers to conduct, evaluate and present effective design-science research. These guidelines address design as: (1) the production of a viable artefact, in the form of a construct, model, method or instantiation; (2) problem relevance, the domain where the artefact is purposeful; (3) design evaluation, the demonstration of the utility and efficacy of the design artefact via well-executed evaluation methods; (4) research contributions that the designed artefact, foundations or methodologies used provides to the community; (5) research rigour, via the application of rigorous methods in both the construction and evaluation of the design artefact; (6) the design as a search process to reach the desired ends while accounting to real-world constraints; and (7) the communication of the research.

Since the objectives of this thesis are synthetic (i.e. the development of a framework to support the dynamic evolution of systems), a research methodology consistent with the principles of design science has been followed. Next, through the lens of the design-science guidelines presented by Hevner et al. (Hevner et al., 2004), the research performed in this thesis is presented.

The artefact that has been designed is a framework to support dynamic reconfiguration and evolution. The domains where the artefact (i.e. the framework) is purposeful are the areas of self-managed software systems, evolvable systems, and highly-available but also flexible software systems. These domains are emerging, as it can be observed from the growing interest coming from the fields of software architectures, autonomic computing, organic computing, dynamic product lines, self-managed systems, self-adaptive systems, etc. As a result of the novelty of the domains, the available knowledge and theories are still insufficient, so innovative designs are required.

The evaluation of the design artefact has been performed through the implementation of a case study. Case studies are frequently used to describe, understand, and explain a research subject (Yin, 2002), (Easterbrook et al., 2008), and in some cases, to validate its correctness and how precisely the research goals are met. The implementation of a case study has provided a better understanding of the problem and feedback to improve the quality of the design artefact. In addition, the case study has also helped to explain the contributions of the designed artefact when compared to existing practices, and also to disseminate the results of the research.

1.4 Research Hypothesis

The hypothesis of this research is that:

> *Dynamic Reconfiguration and Dynamic Evolution of Types are complementary and should be combined to develop highly dynamic software architectures: reconfiguration for structural changes and type evolution for behavioural changes. This combination will benefit architecture-centric systems with a high degree of: (i) plasticity, which will make their structure malleable at runtime (within some constraints), and (ii) flexibility, which will make their types modifiable at runtime. In addition, due to this complementary nature, they will share some change management mechanisms, such as the safe stopping of the running elements.*

However, the combination of dynamic reconfiguration and type evolution should be realised in a way that their integration in a software system is transparent and do not affect the functional concerns of the system. To address this problem, concepts from the aspect-oriented community were considered. Aspects (Kiczales et al., 1997) are software artefacts designed explicitly to encapsulate the properties and behaviour of the concerns (e.g. distributed communication, persistence, logging, etc.) that crosscut with the main system behaviour (i.e. the functional concerns). In this way, aspects centralize the maintenance and reuse of crosscutting concerns. Then, the starting question was: Why do not encapsulate the behaviour related to dynamic evolution and reconfiguration into aspects? Evolution can be considered as a concern that crosscuts with the system functionality, and then, be encapsulated in an aspect. For this reason, the previous hypothesis was extended with the following statement:

The encapsulation of these evolution concerns into aspects will make their integration in the software system transparent, as well as it will make easy their maintenance.

For the construction of dynamic software architectures, this approach has been based on PRISMA (Pérez, 2006). PRISMA has been selected among other Architecture Description Languages because of the combination it does of Aspect-Oriented Software Development (Filman et al., 2004) and Component-Based Software Development (Szyperski, 2002) for describing complex software architectures. For describing the behaviour of architectural elements, PRISMA provides an aspect-oriented model that has the advantage that explicitly encourages the separation among the different concerns: an architectural element is entirely defined by the combination of different aspects. In addition, PRISMA follows a Model-Driven Development approach, which allows the automatic generation of the final code from high-level (architecture-based) models. In this way, the integration of the evolution concerns in the PRISMA model will allow us to model dynamic software architectures and automatically generate the final code supporting these features. We have called the resulting combination as *Dynamic PRISMA*.

1.5 Thesis Overview

The remainder of this thesis develops as follows:

Chapter 2: Context. This chapter presents an overview of the context of this thesis: Software Architectures, Aspect-Oriented Software Development, and the PRISMA model.

Chapter 3: Dynamic Software Evolution. This chapter presents the main concepts related to dynamic evolution: the different definitions, kinds of evolutions, mechanisms for safe stopping and state transfer, and dynamism in software architectures.

Chapter 4: Related Works. This chapter presents the state-of-the-art related to dynamic evolution and reconfiguration in software architectures.

Chapter 5: Case Study: Agrobot. This chapter introduces the case study that has been chosen to illustrate the approach presented in this thesis: *Agrobot*. This case study is specified in terms of the PRISMA ADL. This chapter also presents how dynamic evolution and reconfiguration can advantage the development of autonomous robotic systems.

Chapter 6: Autonomic Reconfiguration. This chapter describes the elements that have been defined to support the autonomic reconfiguration of hierarchical software architectures. These elements allow a subsystem to autonomously change its architecture at runtime.

Chapter 7: Dynamic Evolution of Architectural Types. Here is described the reflective infrastructure that supports the dynamic evolution of architectural types and instances in an asynchronous way. This allows us to entirely modify the structure and behaviour of architectural elements at runtime, and thus develop a real *new* architecture at runtime.

Chapter 8: Description of the Evolution Semantics. This chapter presents how the asynchronous evolution semantics proposed in this thesis has been described by means of typed graph transformations.

Chapter 9: Conclusions. This chapter presents the main contributions of the thesis, the publication results, and the future research work.

Appendix A: PRISMA Specifications of the VisionSystem. This appendix presents the complete PRISMA specification of a subsystem of the Agrobot, the *Vision System*, as well as the PRISMA specification of the elements that support dynamic reconfiguration and type evolution.

Appendix B: Extensions of the PRISMA AOADL. This appendix presents some language constructs that have been added to the PRISMA Aspect-Oriented Architecture Description Language, to support the features introduced in this thesis.

CHAPTER II

CONTEXT

2.1 Introduction

This chapter presents an overview of the context of this thesis: Software Architectures, Aspect-Oriented Software Development, and the PRISMA approach.

On the one hand, Software Architectures is a discipline that provides techniques for (i) describing the structure (or architecture) of complex software systems (i.e. the key system elements and their organization), and (ii) reflecting the rationale behind the system design. This discipline has been acknowledged as a centric issue in the development of large systems, since it contributes to *programming-in-the-large* rather than to *programming-in-the-small* (DeRemer & Kron, 1976). In addition, the study of software architecture has been justified as the primary way to deal with the evolution and customization of systems (Perry & Wolf, 1992). This thesis focuses particularly on "evolving-in-the-large", as the modification of parts of a running system without shutting it down. For this reason, an introductory view to the discipline of Software Architectures is needed.

On the other hand, Aspect-Oriented Software Development (AOSD) focuses on the study of the crosscutting-concerns of a system and on the ways to improve their modularisation and maintenance, according to the principle of Separation of Concerns introduced by Dijkstra (Dijkstra, 1974). The main idea pursued by this discipline is the separation of the crosscutting-concerns of the system and their automatic recombination or weaving (Elrad et al., 2001). This improves the maintenance of code, simplifies system specifications and allows us to easily extend systems by adding aspects to existing code.

Finally, PRISMA is an approach that integrates Software Architectures and AOSD disciplines to specify software architectures of complex systems. Since this thesis has been materialized using the PRISMA approach, the objective of this chapter is to provide the reader with an introduction to PRISMA in order to permit the comprehension of the coming chapters of this thesis.

This chapter is organized as follows: Section 2.2 presents an overview of Software Architectures. Next, section 2.3 introduces the main concepts related to AOSD. Finally, section 2.4 describes the PRISMA model: its main benefits, its architectural elements, and its model-driven development support.

2.2 Software Architectures

The increasing size and complexity of current software systems has led the computing community to acknowledge the importance of the system's structure as a centric issue in the entire lifecycle of a software system. In their seminal works, Dewayne E. Perry and Alexander L. Wolf (Perry & Wolf, 1992) introduced the need for studying the architecture of software systems. This need was justified as the primary way to deal with two factors that contribute to the high costs of software: the evolution and customization of systems.

A software system without an appropriate architectural design is more difficult to evolve and customize. The architecture of a system tells us what design decisions guided the building of the system, i.e. which architectural elements, interactions and constraints were used for its development. If such design decisions are not taken into account when changing or customizing a system, the system can suffer from *architectural drift* and *architectural erosion*, two phenomena which lead to the fragility and resistance to change of a system (Perry & Wolf, 1992). **Architectural drift** refers to the engineer's insensitivity to the system's architecture and can lead to a loss of clarity of form and system understanding. It may involve decisions whose implications are not properly understood and which may affect the given system's future adaptability (Taylor et al., 2009). On the other hand, **architectural erosion** refers to the introduction of architectural design decisions that violate the system prescriptive architecture. It can easily occur when a system has been drifted too far, as a consequence of many small, intermediate changes that obscure violations on important architectural decisions (Taylor et al., 2009). Both architectural drift and architectural erosion can be dangerous and expensive, and should be avoided. This was one of the factors that motivated the emergence of Software Architecture.

Software Architecture was presented as a solution for the design and development of large, complex software systems. The reason is that architecture allows us to describe the structure of a software system by hiding the low-level details and abstracting the high level important features, thus making software systems simpler and more understandable (Garlan & Perry, 1995). Other authors contributed to defining and establishing the foundations of the Software Architecture discipline. One of the earliest books on the matter was the book of Mary Shaw and David Garlan (Shaw & Garlan, 1996), which provided a collection of definitions, summaries of industrial and research projects and early architectural insights from those projects. Other interesting books focused on software architecture patterns (Buschmann et al., 1996), architecture modelling (Hofmeister et al., 1999), and architecture evaluation (Clements et al., 2002). Over the years, software architecture has been consolidated as a focus of software engineering research (Shaw & Clements, 2006), (Taylor & Hoek, 2007), and is becoming a centric element of the entire development phase (Taylor et al., 2009).

2.2.1 Definition

In the last decade, several definitions about software architecture have been proposed. Next we introduce and explain those that consider the most relevant. The first definition is the proposed by Perry & Wolf in 1992:

Software Architecture = {Elements, Form, Rationale}

> Elements *capture the system's building blocks, which can be of three types: processing elements, data elements and connecting elements.*
>
> Form *captures how the (architectural) elements are organized in the architecture, by means of weighted properties and relationships. That is, the form captures how the elements are composed (i.e. the architecture configuration), the characteristics of their interactions, and their relationship with their operating environment.*
>
> Rationale *captures the motivation for the choice of an architectural style, the choice of elements, and the form. That is, the system designer's intent, assumptions, choices, external constraints, selected design patterns, and other information that is not easily observable from the architecture.*
>
> *(Perry & Wolf, 1992)*

This definition is one of the most widely extended and accepted definitions of Software Architecture. It is interesting the practical characterization of the previous definition provided by Taylor, Medvidovic and Dashofy, by means of *What*, *How* and *Why* questions:

> Elements *help to answer the What questions about the architecture: What are the elements of a system? What are their primary purpose and the services that they provide?*
>
> Form *helps to answer the How questions about the architecture: How is the architecture organized? How are the elements composed to accomplish the system's key task? How are the elements distributed?*
>
> Rationale *helps to answer the Why questions about the architecture: Why are particular elements used? Why are they combined in a particular way? Why is the system distributed in a given manner?*
>
> *(Taylor et al., 2009)*

Another extended definition was the proposed by Mary Shaw and David Garlan, which implicitly includes the elements defined in the definition of Perry and Wolf (i.e. elements, organization and rationale):

> *Software architecture [is a level of design that] involves the description of elements from which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.*
>
> *(Shaw & Garlan, 1996)*

In addition, there is another definition provided by the *ANSI/IEEE Standard 1471-2000*:

> *Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.*
>
> *(IEEE, 2000)*

Note that this definition is similar to the previous ones, but it explicitly addresses system evolution, and it does not specifically refer to software.

Finally, one of the most recent definitions is the provided by Taylor, Medvidovic and Dashofy in his book about software architecture:

> *A software system's* architecture *is the set of principal design decisions made about the system.*
>
> Design decisions *encompass every aspect of the system under development, including: system structure, functional behaviour, interaction, nonfunctional properties and implementation.*

> Principal *is a term that implies a degree of importance and topicality that grants a design decision architectural status, that is, that makes it an architectural design decision (i.e. it impacts a system's architecture).*
>
> *(Taylor et al., 2009)*

This definition is more abstract than the previous definitions, and explicitly gives the system architecture specification with a central role in the development of a software system. Therefore, architecture do not only comprises the structure of a system, but the description of the main functional behaviour (e.g. data processing, storage, visualization, databases, …), the kind of interactions that will be implemented (e.g. event-based communication, procedure-based communication, etc.), the non-functional properties (e.g. dependability) and the implementation technology (e.g. Java, .NET 3.5, …).

It can be noticed that each definition presents different issues. However, it can be concluded that each definition is mainly concerned with structure and behaviour. Structure describes how the system is made up of interconnected units called components. Behaviour is referred to the visible behaviour caused by the interaction of the systems components to achieve the overall functionality of the system. Structure and behaviour is formally specified using an Architecture Description Language (ADL), which is used later to build the executable code of the software system.

2.2.2 Basic Concepts

Despite the wide set of ADLs defined up to date (Medvidovic & Taylor, 2000), there are some concepts that are common to all of them. In order to facilitate a better comprehension of the work of this thesis, the most relevant concepts are presented here.

2.2.2.1 Component

The concept of component is the basis of software architecture and the concept that Architecture Description Languages (ADLs) share par excellence:

> *A software component is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context.*
>
> *(Taylor et al., 2009)*

A component is a computational element that permits users to structure the functionality of software systems. It has a high level of encapsulation and it is only possible to interact with it by means of its interfaces. Thus, components are considered as black boxes, which embody the software engineering principles of encapsulation, abstraction and modularity. A component can be as simple as a single operation or class, or as complex as an entire system, depending on the architecture, the perspective of the designers and the needs of the system.

2.2.2.2 Connector

Connectors describe the interactions among components:

> *A software connector is an architectural element tasked with effecting and regulating interactions among components.*
>
> *(Taylor et al., 2009)*

Connectors were first introduced by Mary Shaw to explicitly separate computation from coordination and improve the separation among these concerns:

> *Connectors are the locus of relations among components. They mediate interactions but are not "things" to be hooked up (they are, rather, the hookers-up). Each connector has a protocol specification that defines its properties. These properties include rules about the types of interfaces it is able to mediate for, assurances about properties of the interaction, rules about the order in which things happen, and commitments about the interaction such as ordering, performance, etc.*
>
> *(Shaw, 1994)*

In her work, she presents the need for connectors due to the fact that the specification of software systems with complex coordination protocols is very difficult without the notion of connector. From her experience in the software architecture field, she demonstrates that the connector provides not only a high level of abstraction and modularity to software architectures, but also an architectural view of the system instead of the object oriented view of compositional approaches. She also defends the idea of considering connectors as first-class citizens of ADLs.

This idea is also emphasized in subsequent works of other authors, such as in (Allen & Garlan, 1997). In fact, connectors imply a runtime mechanism for transferring control and data around a system (Bas et al., 2003). Examples of connectors are procedure call, event broadcast, and pipes. In this way, since components only deal with functionality and data instead of interactions, connectors indirectly benefit the reusability and modularity of components.

Connectors do not simply redirect calls among two components (as method invocations). Moreover, connectors can provide and encapsulate services such as persistence, invocation, messaging, and transactions. These kinds of services are usually considered as part of "facility components" in middlewares such as CORBA, DCOM or RMI. However, by considering such services as connectors, it helps to clarify an architecture and keep the component's focus on application and domain-specific concerns.

Taylor et al. (Taylor et al., 2009; chapter 5) have identified four classes of services a connector can provide:

- *Communication*, i.e. transmission of data among components;
- *Coordination*, i.e. transfer of control among components;
- *Conversion*, i.e. transform the interaction required by one component to that provided by another;
- *Facilitation*, i.e. services that mediate and streamline component interaction. For instance, load balancing, scheduling services or concurrency control.

Every connector provides services that belong to at least one of these four categories. In the literature, several types of connectors have been defined, which may provide multiple services. For instance, procedure call provides both communication and coordination services. Other types of connectors are: event, data access, linkage, stream, arbitrator, adaptor, and distributor. For further information, an excellent study of the different types of connectors and their properties for building a software architecture can be found in (Taylor et al., 2009; chapter 5).

2.2.2.3 Configuration

Components and connectors are composed in a specific way to accomplish the system's objectives. This composition represents the system's configuration, also referred to as topology:

> *An architectural configuration is a set of specific associations between the components and connectors of a software system's architecture.*
>
> *(Taylor et al., 2009)*

That is, a configuration is a specific structure for a concrete system. In some formalizations, configurations are generally represented as a graph wherein nodes represent components and connectors, and whose edges represent their associations (topology or interconnectivity): (Hirsch et al., 1998), (Wermelinger et al., 2001). The associations among components and connectors are sometimes called attachments.

2.2.2.4 System

It is frequently the case that different abstraction levels must be provided to facilitate the understandability and the specification of the architectural description. For this reason, mechanisms to describe architectural elements with different granularity levels are always desirable. These needs have led to a wide variety of architectural models to introduce the concept of *System* as a composite component, i.e. a component that is made up of other architectural elements. An example is the model proposed by Ivar Jacobson (Jacobson et al., 1997) that introduces the concept of subsystem as a set of organized components.

Systems represent architectural configurations that are made up of connectors and components that can be built in a hierarchical way. For this reason, a system can be composed of other subsystems (Andrade & Fiadeiro, 2003). Other ADLs have also introduced the concept of composite component or system, such as Darwin (Magee et al., 1995), ArchWare ADL (Oquendo et al., 2004) or ACME (Garlan et al., 2000).

The concept of system differs from configuration in that a system is a building block (i.e. another kind of architectural element) that can be reused in several software systems, whereas a configuration defines the structure of a specific system (i.e. it cannot be reused).

2.2.2.5 Port

The concept of port is related to architectural elements (i.e. components and connectors). Ports are the points of interaction through which architectural elements can interact with the other elements of a software architecture. They are the parts into which the interface of an architectural element is divided. Their main function is to preserve the black box view of architectural elements and to publish the behaviour offered and required by architectural elements. They have been used in different ways; some approaches consider a port as a service and other approaches as a process with several services. This last way of defining ports, not only defines the services of ports, but also the conditions of how and when they can be required and provided.

2.2.2.6 Connection

Connections are used to constrain the “placement" of architectural elements. That is, they constrain how the different architectural elements may interact and how they are organized with respect to each other in the architecture (Perry & Wolf, 1992). Connections attach two ports of architectural elements, generally a component port and a connector port in ADLs where connectors are considered first-class citizens. These connections are usually called *attachments*.

2.2.2.7 Compositional Relationship

Compositional relationships emerge along with the concept of systems, due to the fact that it is necessary for systems to communicate with their constituent architectural elements. These connections are different from attachments because they are used to connect architectural elements of different levels of granularity. As a result, the semantics of these connections is compositional, whereas attachments have a communication semantics that is not compositional (the same level of granularity). These relationships are usually called *bindings*.

Bindings establish the mappings between the internal and external interfaces of a system (Garlan, 2001). As a result, bindings establish a connection between a system port and a port of one of its architectural elements.

2.2.2.8 Other concepts

There are other relevant concepts related to Software Architecture that remain to be defined. For instance, the concept of Architecture Style (Perry & Wolf, 1992), (Abowd et al., 1993) is relevant to define general design decisions about the architectural elements and to emphasize important constraints on the elements and its relationships. Architectural styles are used to represent families of software architecture descriptions that belong to software systems that have something in common: resource types, configuration patterns and constraints (Garlan, 2001). Example of styles are event-based, publish-suscribe, blackboard, pipe-and-filter, client-server, object-oriented, etc. It is also relevant the concepts of property (Garlan, 2001) and Constrainte (Andrade & Fiadeiro, 2003) to describe the semantics associated to architectural elements or the restriction of the design, respectively. The reader can refer to (Taylor et al., 2009) to get further details about all the relevant topics within software architecture.

2.3 Aspect-Oriented Software Development

Since the "software crisis" emerged in the late 1960s, several efforts have dealt with the management of complexity of the software development process. Parnas proposed the term *modularization* as a criterion to simplify software development and improve software understanding and quality (Parnas, 1972). Modularization decomposes complex software systems into smaller parts called modules. Modularity helps when developing systems, since modules can be replaced by other modules while the rest of the system remains intact.

However, system designers would like to modularly exchange in several dimensions: different features should be exchanged independently of each other. An illustrative example of this is provided in (Aßmann, 2003) through an analogy. For instance, in the architecture of buildings, plans for rooms, water, gas and electricty are specified separately. When architects want to exchange parts of the electricity support for a room, they never exchange the complete room. Instead, they only modify the electricity plan of the room, which does not affect the other plans. After all plans are finished, the construction process integrates them into the physical layout of the building and eliminates remaining conflicts.

The practice of dividing software into different areas of interest is widely referred to as *Separation of Concerns* (SoC) (Dijkstra, 1974). Separation of concerns is the subdivision of a problem into independent parts. For instance, an example of separation of concerns in software architecture is the separation of a system's structure into components (loci of computation) and connectors (loci of communication). However, this is not enough, since several concerns (such as persistence, audit, logging, security, etc.) may remain intertwined across different components. As (Jacobson & Ng, 2003) set out, components are developed to satisfy several requirements, which often lead to a development where the concerns of the systems are *tangled*[1] and *scattered*[2] across the architecture of the system.

2.3.1 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) (Kiczales et al., 1997) (Elrad et al., 2001) emerged as another approach for realizing SoC, but showing a key difference: it focuses on those concerns that crosscut a software system, facilitating that each concern can be separately specified.

Therefere, the notion of *concern* is close to that provided by the IEEE standard 1471-2000:

[1] *Tangling*: the material pertaining to multiple requirements is interleaved within a single module (Tarr et al., 1999). As a result the module is less maintainable, reusable and comprehensible.

[2] *Scattering*: a single requirement affects multiple design and code modules (Tarr et al., 1999). This results in identical definitions repeated in multiple modules.

...those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders.

(IEEE, 2000)

The problem that AOP tries to solve is how to manage properly those catalogued as *crosscutting-concerns*, i.e. concerns that are scattered and tangled. Software systems are usually crosscut by common concerns of a domain system, and these crosscutting-concerns are spread throughout the software units of the system. As a result, the crosscutting-concerns are repeated in all the software units that they affect, and these concerns are tangled with the other concerns that also modify the same software unit. This increases the volume of code and complicates the maintenance that preserves the consistency of changes. In addition, tangled concerns make the maintenance of a specific concern more costly because it is so difficult to locate the correct place to introduce changes.

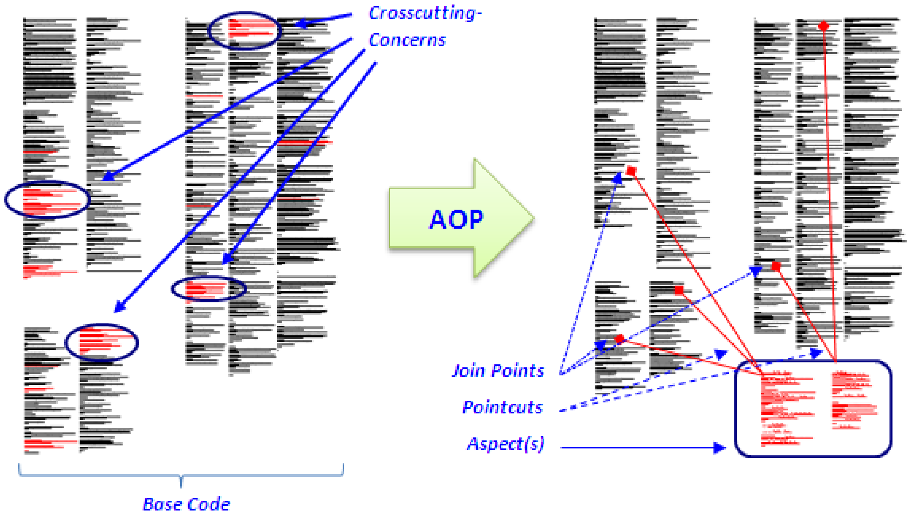


Figure 2.1. Advantages for Modularity with AOP

As a solution to this problem, AOP proposes the separation of the crosscutting-concerns of software systems into separate entities, which are called *aspects*. Aspects are software units that can be reused throughout the system by weaving them wherever necessary (classes, modules, components, etc.), managing properly the tangled and scattered code. Kiczales (Kiczales et al., 1997) demonstrated that the use of aspects means advantages in terms of understandability of code, maintainability and reusability. Figure 2.1 shows graphically how the grouping of the crosscutting-concerns of a system into isolated units, *aspects*, benefits the modularity of code. AOP promotes the idea that software systems are better programmed using the notion of aspect and considering aspects as first-class citizens of programming languages. AOP does not only offer aspects to encapsulate crosscutting-concerns, but it also provides mechanisms to weave aspects with the rest of the software system.

2.3.2 Basic Concepts

Next, the essential notions of AOP are presented, in order to understand how aspects encapsulate crosscutting-concerns and are weaved to the system:

- **Base code** is that code that describes the core functionality of a program or domain, where the aspects are woven. Every aspect encapsulates a crosscutting concern for the specific program or domain. The *aspect code* collects the set of defined aspects.
- **Weaving** is the process that combines the concerns of the system, i.e. the integration of the base code and the aspectual code. Weaving rules are specified outside the base code to make it unaware of the woven aspects. Thus, the system can be changed only by changing the weaving rules.
- **Join points** are well-defined points in the structure of a program (i.e. the base code) where aspect code can be attached or hooked. The most common elements of a join point are method calls. These points can be extended with aspect code, thus changing the original flow of control of the system.
- **Pointcut** is a set of join points which have been selected for injecting aspect code at run-time. When the execution reaches one of them, the advice (a piece of code) is executed to determine the sequence of execution at that point. A pointcut allows the aspect to do something with a single statement in many places.
- **Advice** is the behaviour to be executed at a join point that has been selected in a pointcut. An advice determines if the aspect must be executed *before*, *after* or *instead* the code of the join point.
- **Aspect** is a language constructor that encapsulates a crosscutting-concern, and is composed of pointcuts and one or more advices. Aspects can have their own state.

Among the different approaches and implementations of AOP, two weaving models are distinguished: *static weaving* and *dynamic weaving*. On the one hand, *static weaving models* are those in which the aspects and non-aspect entities are declared as separate entities, but at compilation time the two entities are combined into one. This model has the drawback that at execution time the aspects cannot be manipulated. As a result, aspects fail to gain a high level of evolution, maintenance and reusability. Examples of this model are AspectJ (Kiczales et al., 2001), Composition Filters (Aksit et al., 1994) or Loom.NET (Schult & Polze, 2002). On the other hand, dynamic weaving models are those in which the separation of aspects and non-aspect entities is conserved at all moments even at execution time. Aspects can be woven and unwoven at run-time. This benefits the evolution and adaptation of systems, since this model allows us to extend existing code by adding or removing aspects. Examples to this model are the Disguises Model (Sanchez et al., 1998), PROSE (Popovici et al., 2002), JAsCo (Suvée et al., 2003), EOS (Rajan & Sullivan, 2003), and the Dynamic Aspect-Oriented middleware Framework (DAOF) (Pinto et al., 2005).

2.3.3 Aspects in the Software Life Cycle

Crosscutting-concerns do not only arise in the development phase of the software life cycle, but in *every phase* of the software life cycle. For this reason, although AOP emerged from the implementation level, its use has been extended to all the stages of the software life cycle: such as requirements (Navarro, 2008) or architecture (Pérez, 2006). This is how the term Aspect-Oriented Software Development (AOSD) (Filman et al., 2004) has emerged exploiting the advantages this paradigm can provide in every stage of software development. For further details about AOSD and the different existing approaches and models, the reader can refer to (Chitchyan et al., 2005), (Douence & Le Botlan, 2005), or (Pérez, 2006; chapter 3), where an extensive introduction about the AOSD field is presented.

2.4 PRISMA

PRISMA provides a model for the definition of complex software systems (Pérez et al., 2005b). Its main contributions are the way in which it integrates elements from aspect-oriented software development and software architecture approaches, as well as the advantages that this integration provides to software development. The PRISMA model introduces the notion of aspect following an architectural model with connectors and a symmetrical aspect-oriented model.

Since the PRISMA model is a technology-independent model, the PRISMA approach also follows the Model-Driven Development paradigm (Selic, 2003) to obtain its advantages during the development and maintenance processes of PRISMA architectures. The main goal of the PRISMA approach is to give a complete support for the development of technology-independent aspect-oriented software architectures, which could be compiled to different technological platforms and languages using automatic code generation techniques.

The purpose of this section is to present the main properties of the PRISMA model and the reasons why it has been selected as the framework for developing the ideas of this thesis.

2.4.1 Model and ADL selection

Among the different formal Architecture Description Languages (ADLs) from the literature (Medvidovic & Taylor, 2000), we have selected the PRISMA ADL (Pérez, 2006) because of the advantages it provides for supporting dynamic evolution of software architectures.

First, the PRISMA language allows modelling (Pérez et al., 2006), (Pérez & Cuesta, 2007): (i) the functional decomposition of a system, by using architectural elements; and (ii) the system's crosscutting concerns, by using aspects. This results in simpler, clearer, and more concise system specifications. In addition, this allows us to separate those parts of the software that exhibit different rates of change, and evolve only the interesting parts (Mens & Wermelinger, 2002). In this way, we can easily isolate functional and reconfiguration concerns.

Second, PRISMA does not only allow modelling the structure (i.e. the architecture) of a system, but also allows describing precisely the internal behaviour of each architectural element. The behaviour is specified by using: (i) a modal logic of actions for describing services (Stirling, 1992); and (ii) π-calculus with priorities, for describing interactions among services (Milner, 1993). Thus, since the internal behaviour is formally described, this allows us to automatically interleave the actions required to perform the runtime evolution of its instances: (i) actions to achieve quiescence, and (ii) actions to perform the state migration.

Lastly, the PRISMA ADL is supported by a Model-Driven Development framework, PRISMACASE (Pérez et al., 2008), which allows the automatic generation of executable code from PRISMA models/specifications. This also benefits the support for dynamic evolution. The code generation templates can include not only the code for supporting the runtime evolution of the system, but also the code to reflect the changes on the formal system specification, keeping both in sync. For this reason, next we introduce the main concepts of the PRISMA ADL.

2.4.2 Aspects as First-Class Citizens

Aspect-Oriented Software Development allows us to encapsulate in a single artefact (the aspect) the behaviour that is disseminated across a system (i.e. a crosscutting concern).

One of the contributions that the PRISMA model provides to software architecture is that it defines an Architecture Description Language that introduces aspects as a new concept of software architectures (Pérez et al., 2006), rather than simulating them using other existing architectural terms (components, connectors, views, etc). Then, aspects are first-class citizens of software architectures and represent a specific behaviour of a *concern* (safety, coordination, distribution, reconfiguration, etc.) that crosscuts the software architecture. Moreover, one concern can be specified by several aspects. Aspects can be reused: the same aspect can be imported by each one of the architectural elements (components and connectors) that need to take into account the behaviour of the concern that this aspect defines.

Another contribution of the PRISMA model is that the behaviour of architectural crosscutting concerns is defined by means of a symmetrical aspect-oriented model (Harrison et al., 2002), (Cuesta et al., 2005). In this kind of models, aspects are not constrained to only specify non-functional requirements; aspects also specify functional requirements (i.e. the business logic). As a result, PRISMA provides a homogeneous treatment for functional and non-functional requirements.

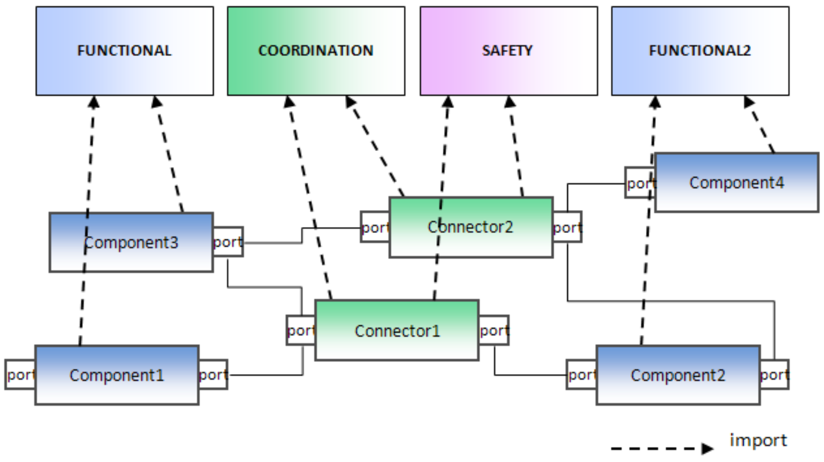


Figure 2.2. Crosscutting-concerns in PRISMA architectures

Thus, the behaviour of a PRISMA architectural element is defined by a set of aspects, which describe the architectural element from different *concerns* of the architecture and can be reused by other architectural elements at the same time (see Figure 2.2). In this sense, aspects crosscut those elements of the architecture that import their behaviour.

Since PRISMA is a symmetrical aspect-oriented model that it is applied to software architectures, i.e. the architectural level, the weaving process does not define pointcuts between base code and aspect code and their corresponding advices. In PRISMA, there is no base code; all system behaviour is defined as an aspect. As a result, the weaving process is composed of a set of *weavings*, and a weaving indicates that the execution of an aspect service can trigger the execution of services in other aspects.

From the AOP point of view (see section 2.3), PRISMA weavings can be defined as follows: every service of an aspect is a *join point*, the services that trigger a weaving are the *pointcuts*, and the services that are executed as a consequence of weavings are the *advices*. A weaving is defined by means of operators that describe the order in which services are executed. A weaving has the following structure:

<aspect1>.<service1> <weaving_operator> <aspect2>.<service2

Where:

- *<service1>* and *<service2>* are services that are defined in the aspects *<aspect1>* and *<aspect2>*
- *<weaving_operator>* can be one of the following: *after, before, instead, afterIf*(<boolean condition>), *beforeIf* (<boolean condition>), and *insteadIf* (<boolean condition>). These operators define if <service1> is executed *after, before* or instead <service2>.

In PRISMA, to preserve the independence of the aspect specification from other aspects and weavings, weavings are specified outside aspects and inside architectural elements. As a result, aspects are reusable and independent of the context of application and weavings weave the different aspects that form an architectural element. This way of specifying weavings achieves not only the reusability of the aspects in different architectural elements, but also the flexibility of specifying different behaviours of an architectural element by importing the same aspects and defining different weavings.

2.4.3 PRISMA Architectural Elements

PRISMA has two kinds of architectural elements: *simple* (Components and Connectors) and *composite* (Systems[3]). A simple architectural element can be seen as a part of a system which cannot be decomposed into simpler parts. A composite architectural element can be seen as a subsystem, which defines a logical or physical composition of architectural elements. This allows us to increase the modularity, composition and reuse of architectural elements.

Externally, simple and composite architectural elements are similar: both encapsulate their functionality as a black box (see Figure 2.3), which publishes a set of services that they offer to other architectural elements. These services, grouped in interfaces, are provided through *ports*, which are the interaction points among architectural elements. A port can provide server behaviour (i.e. it provides services), client behaviour (i.e. it requires services), or both (i.e. it provides and requires services). The interactions among architectural elements are called *attachments*: an attachment links the port of an architectural element to the port of another architectural element.

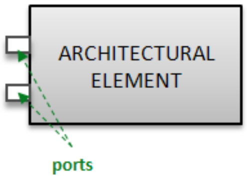


Figure 2.3. Black-box view of PRISMA architectural elements

However, simple and composite architectural elements differ in its internal composition. On the one hand, the internal view of a **simple architectural element** is defined as an *invasive composition* (Aßmann, 2003) of aspects. This can be shown as a prism (see Figure 2.4), where each side of the prism is an aspect that the architectural element imports. An aspect defines the state and behaviour of a specific concern: (i) the state at any given moment is determined by the value of its attributes; and (ii) the behaviour is defined by the semantics of the services that the aspect provides. More details about the semantics of aspects can be found on (Pérez et al., 2006) and (Pérez, 2006). Aspects are synchronised among them by means of weavings. Thus, the behaviour of a simple architectural element *emerges* from the set of aspects it is invasively composed of.

[3] To avoid confusions, we will use capitalized letters when referring to concepts of the PRISMA metamodel: Component and Connector (simple architectural elements), and System (a composite architectural element).

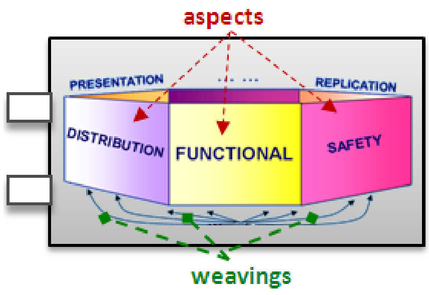


Figure 2.4. Internal view of simple PRISMA elements

There are two kind of simple architectural elements: *Components* and *Connectors*. The difference between a Component and a Connector is that a Component captures the functionality of a software system, whereas a Connector acts as a coordinator among other architectural elements. This difference of roles is reflected in the PRISMA model in the fact that components define a *functional* aspect, whereas connectors define a *coordination* aspect.

On the other hand, the internal view of a **composite architectural element** consists of architectural elements and the links among them (see Figure 2.5). These architectural elements can be of any kind, either simple (i.e. Components and Connectors), and/or composite (i.e. Systems), whereas the links among them can be of two kinds: *attachments* or *bindings*. Attachments allow architectural elements to interact with each other. Bindings are a kind of connection that enables the communication of internal elements with external architectural elements: a binding links the ports of a composite architectural element and an (internal) architectural element.

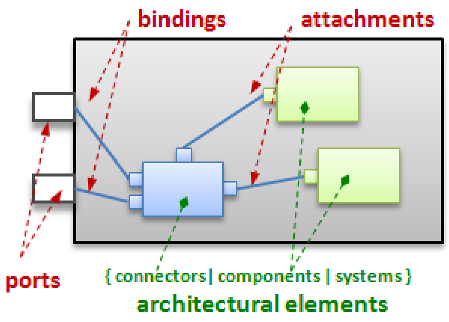


Figure 2.5. Internal view of composite PRISMA elements

Further details about the semantics of the PRISMA ADL can be found in (Pérez et al., 2006) and (Pérez, 2006).

2.4.4 Levels of abstraction

The PRISMA ADL defines the architectural elements of a software system at different levels of abstraction: the *type definition level* and the *configuration level*. The type definition level defines architectural types, which are instantiated in specific architectures or are reused by other architectural types. The configuration level defines the architecture of a concrete software system, by creating and connecting instances of the architectural types defined at the type definition level. In other words, the configuration level specifies the topology of a specific architectural instance. This separation among the type level and the configuration level allows to easily differentiate (and implement) changes in a type, and changes in a configuration (i.e. an *architectural* instance).

An architecture is defined at the type-level as a pattern, so that it can be reused in any other system or architectural type. The architectural element that describes an architecture, and thus defines it through a pattern, is a composite architectural type[4]. A composite type can be used in other architectural types as a single unit, and be treated like other simple architectural types (i.e. Components and Connectors). This allows PRISMA to support the compositionality, or hierarchical composition, of its architectural elements: the architecture of a complex software system can be described as a composition of several architectural elements which, in turn, can be described as the composition of other architectural elements. Thus, a complex system can be recursively defined as an architecture of architectures, because each composition describes an architecture.

The pattern of a composite architectural type (i.e. a System) defines: (i) a set of ports for communicating with its environment (or with other architectural elements); (ii) the set of architectural types it is composed of and the number of instances that can be created of each type; and (iii) the set of valid connections among the architectural types and the number of connections allowed.

For instance, Figure 2.6-top shows a composite type called *Sys*. It consists of two architectural element types, *A* and *B*, with a cardinality of *1..1* and *1..n*, respectively. These types are connected to each other by an Attachment called *Att_AB* with a cardinality *1..1* and *1..n*. These cardinalities mean that only one instance of A is allowed, which can be connected to several instances of B. Finally, *Sys* interacts with its environment by means of the port *p1*. The

[4]In the PRISMA metamodel, a composite architectural element is called *System*. This is the reason why the terms "composite architectural element" and "System" (with a capitalized letter) have been used throughout this thesis as synonyms.

behaviour of this port is provided by the architectural type *A*, which is connected by means of the Binding called *Bin_p1A*.

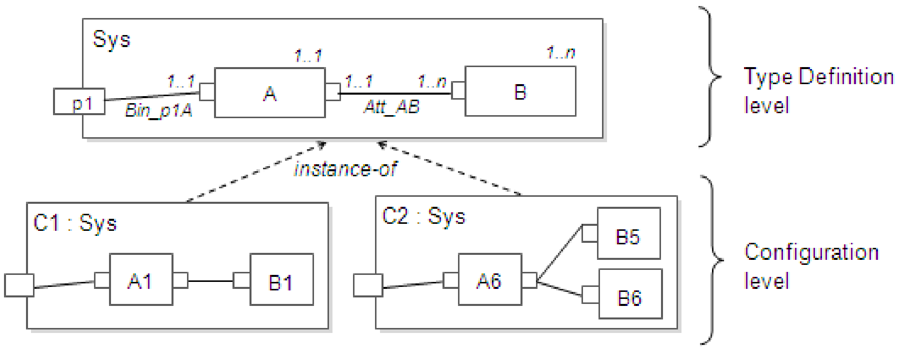


Figure 2.6. Example of a composite type and two possible instantiations

On the other hand, the instantiation of a composite type is defined at the configuration-level, and is called *Configuration* (i.e. a composite instance). A Configuration instantiates a concrete architecture from the different combinations allowed by the pattern: it instantiates each of the architectural types defined in the pattern and connects them appropriately. For instance, Figure 2.6-bottom shows two Configurations, *C1* and *C2*, of the System *Sys*. For illustration purposes, the PRISMA ADL specification of System *Sys* and Configurations *C1* and *C2* is shown respectively in Figure 2.7 and Figure 2.8:

```
System Sys
   Ports
      P1 : interface1;
   End_Ports;

   Import Architectural Elements
      A:TA(1,1), B:TB(1,n);

   Attachments
      Att_AB: A.PServ(1,1) <--> B.PServ(1,n);
   End_Attachments;

   Bindings
      Bin_p1A: P1(1,1) <--> A.PClient(1,1);
   End_Bindings;

   new() { /* Constructor definition */ }
   destroy() { /* Destructor definition */ }
End_System Sys;
```

Figure 2.7. Example of a PRISMA System

The distinction among abstraction levels (type definition level and configuration level) provides important advantages. On the one hand, it allows us to independently manage architectural types and configurations:

architectural types can be reused in several systems, whereas architectural configurations define the specific combination of architectural types which define a system. This improves an easy reuse and maintenance. On the other hand, the distinction among type and configuration level allows us to clearly distinguish among type evolution, which is performed at the type definition level and is spread to all the configurations that use the evolved type (see Chapter 7), and reconfiguration, which is performed at the configuration level and only affects a specific system (see Chapter 6).

```
Architectural_Model_Configuration C1 =
   new Sys {
      A1 = new A();
      B1 = new B();

      att_A1B1 = new Att_AB(A1, B1);
      bin_A1 = new Bin_p1A(A1);
   }

Architectural_Model_Configuration C2 =
   new Sys {
      A6 = new A();
      B5 = new B();
      B6 = new B();

      att_A6-B5 = new Att_AB(A6, B5);
      att_A6-B6 = new Att_AB(A6, B6);
      bin_A6 = new Bin_p1A(A6);
   }
```

Figure 2.8. Example of PRISMA Configurations

2.4.5 Model-Driven Development Support

The PRISMA approach follows the Model-Driven Development (MDD) paradigm (Selic, 2003) (Beydeda et al., 2005). There are two main approaches that apply this paradigm. They are the Model-Driven Architecture (MDA) approach proposed by the OMG (OMG, 2003), and the Software Factories approach proposed by Microsoft (Greenfield et al., 2004). MDA deals with the lack of software system adaptation to different technologies and programming languages by proposing four levels of abstraction: CIM (Computation Independent Model), PIM (Platform Independent Model), PSM (Platform Specific Model), and the final application. Software Factories leads to the reuse of architectures, software components, techniques and tools to improve software development.

PRISMA follows MDD in the general sense, that is, it is not focused on MDA or Software Factories. PRISMA MDD support is not constrained to the definition of a specific number of levels of abstraction or techniques because it can vary depending on the needs of each software system. PRISMA follows the MDD approach by providing the software architect with models, which allow for completely developing aspect-oriented software architectures. Since the level of abstraction of models is higher than programming languages and the code is automatically generated from these models, the tasks of the software architect are facilitated. In addition, the use of code generation techniques improves the development and maintenance processes of software.

2.4.5.1 PRISMA in MOF

In order to present how PRISMA model specifications follow the MDD approach, the OMG Meta-Object Facility (MOF) specification is going to be used (OMG, 2002). MOF allows us to clearly present the differences between types and instances and their correspondent models.

MOF defines a four-level "architecture" and its main purpose is the management of model descriptions at different levels of abstraction and their static modification. The upper layer, M3, is the most abstract one (see the M3 layer, Figure 2.9). This layer defines the abstract language used to describe the next lower layer, which contains metamodels. The MOF specification proposes the MOF Model as the abstract language for defining all kinds of metamodels, such as UML or PRISMA.

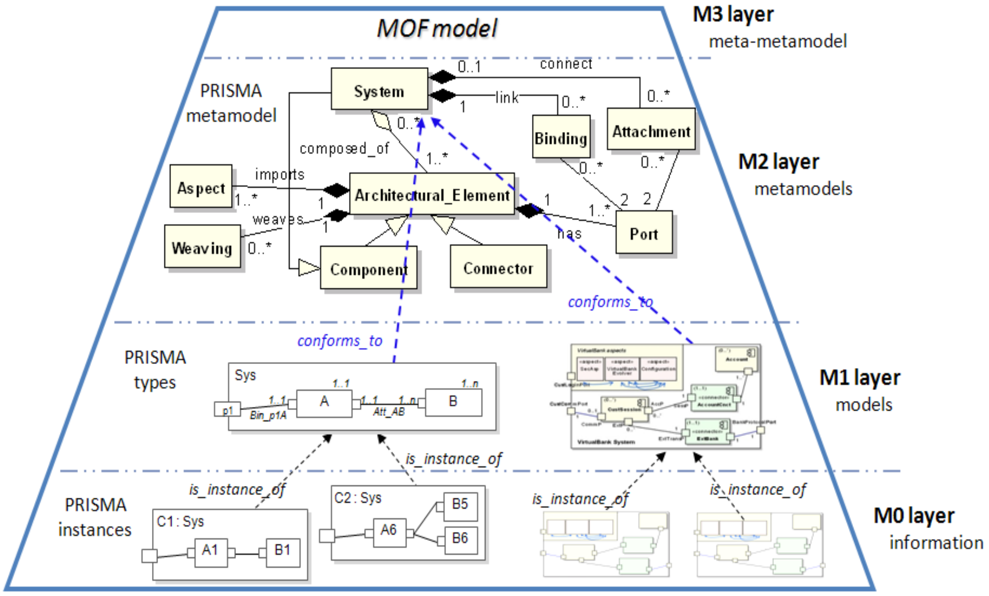


Figure 2.9. Meta-Object Facility layers and PRISMA models

The metamodel layer, M2, defines the structure and semantics of the models defined at the next lower layer. The PRISMA metamodel is defined at this level. It defines the properties that interface, aspect, architectural element, and connection primitives have (see the system package of the PRISMA metamodel in the M2 layer, Figure 2.9).

The M1 layer comprises the models that describe a software system. These models are defined using the primitives and relationships that are described in the metamodel layer (M2). PRISMA models are defined using the interface, aspect, architectural element, and connection primitives that are defined in the previous level (M2). As a result, PRISMA types that are placed in the M1 layer satisfy the properties established at the M2 layer. An example is the *Sys* type, presented in the previous section, which has been defined using the PRISMA system primitive (see M1 layer, Figure 2.9). PRISMA system types are defined as architectural patterns, which are not specifically configured until a particular instantiation is performed.

The lowest level is the information layer (M0 layer), which contains the data, that is, the instances of a specific model. In PRISMA, these data are particular system instantiations (see C1 and C2, M0 layer, Figure 2.9), which behave as described in the system type.

2.4.5.2 The PRISMA MDD Process

The PRISMA model is a metamodel that permits the definition of PRISMA type models whose instantiation defines PRISMA configuration models. PRISMA configuration models define specific systems.

PRISMA applies MDD to define type models from its metamodel (see step A, Figure 2.10), and to define configuration models from type models (see step B, Figure 2.10). In addition, the PRISMA approach has created a set of transformation patterns to transform PRISMA models into its Aspect-Oriented ADL specifications and into C# code (see steps 1 and 2, Figure 2.10). PRISMA applies these transformation patterns during the development process in order to automatically generate applications from its PRISMA architectural models and to show the formal specification of its models.

This MDD process, together with the models and the generation patterns, are provided by a tool, PRISMA CASE. This tool supports the PRISMA approach and is presented in detail in (Pérez et al., 2006), (Pérez et al., 2007a), (Guillén-Martín, 2007). PRISMA CASE currently supports the generation of C# code that is executable on .NET technology from its aspect-oriented architectural models. The PRISMA CASE is composed of the PRISMA metamodel, a graphical modelling tool, a model compiler, a middleware and a generic graphical user interface to execute the generated code (see Figure 2.11).

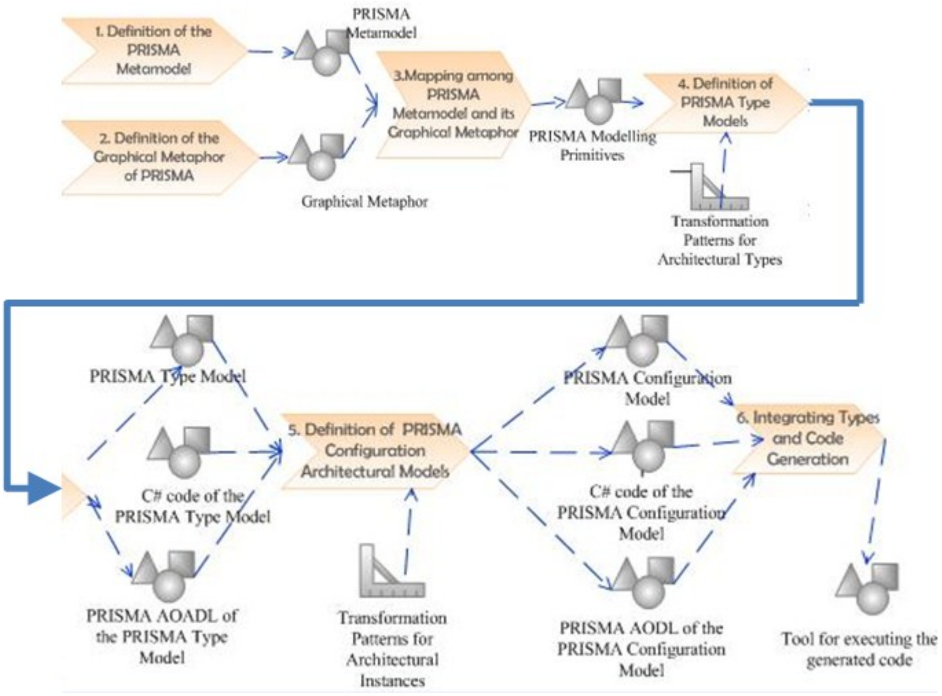

1. Definition of the PRISMA Metamodel
PRISMA Metamodel
2. Definition of the Graphical Metaphor of PRISMA
Graphical Metaphor
3.Mapping among PRISMA Metamodel and its Graphical Metaphor
PRISMA Modelling Primitives
4. Definition of PRISMA Type Models
Transformation Patterns for Architectural Types
PRISMA Type Model
C# code of the PRISMA Type Model
PRISMA AOADL of the PRISMA Type Model
5. Definition of PRISMA Configuration Architectural Models
Transformation Patterns for Architectural Instances
PRISMA Configuration Model
C# code of the PRISMA Configuration Model
PRISMA AODL of the PRISMA Configuration Model
6. Integrating Types and Code Generation
Tool for executing the generated code

Figure 2.10. MDD from the PRISMA Metamodel to Applications

The PRISMA metamodel is part of the PRISMA CASE since the metaclasses that allow the creation of PRISMA aspect-oriented software architectures, as well the constraints of the PRISMA metamodel, must be available in the CASE tool. They are necessary to be able to model PRISMA architectural models and to make sure that they satisfy the PRISMA constraints.

The PRISMA Aspect-Oriented ADL is a formal language. Even though the use of a formal language clearly provides advantageous characteristics, the use of a formal language is really difficult. For this reason, PRISMA CASE provides a graphical language and a graphical modelling tool to model PRISMA software architectures using an intuitive and friendly graphical interface. This PRISMA graphical modelling tool is divided into two modelling tools following the MDD process presented in the previous section: the *PRISMA Type Modelling Tool* and the *PRISMA Configuration Modelling tool*.

Since PRISMA CASE must generate executable C# code in .NET technology and the .NET framework does not provide support for the Aspect-Oriented approach, a PRISMANET middleware has been developed to provide a solution (Pérez et al., 2005a). PRISMANET extends the .NET technology through the execution of aspects on the .NET platform in accordance with the PRISMA model.

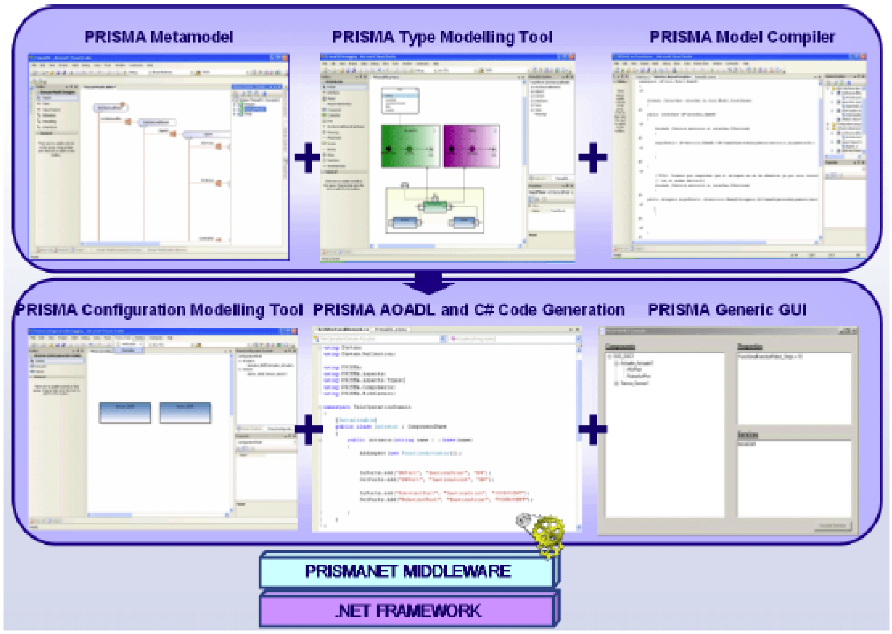

PRISMA Metamodel
PRISMA Type Modelling Tool
PRISMA Model Compiler
PRISMA Configuration Modelling Tool
PRISMA AOADL and C# Code Generation
PRISMA Generic GUI
PRISMANET MIDDLEWARE
.NET FRAMEWORK

Figure 2.11. PRISMA CASE (Perez, 2006)

Finally, the PRISMA model compiler has been developed to automatically generate PRISMA AOADL specifications and C# code from the PRISMA architectural models, and a generic GUI is provided to assist the user in checking the behaviour of the architecture.

2.5 Conclusions

This chapter has presented a brief overview about Software Architecture and Aspect-Oriented Software Development. The most important concepts have been described to facilitate the comprehension of the remaining chapters of this thesis.

In addition, this chapter has introduced the PRISMA approach, which allows the modelling of aspect-oriented software architectures and their code generation. The advantages of PRISMA have been analysed and emphasized to justify the reasons why PRISMA has been selected to apply the contributions of this thesis. Thereby, since PRISMA does not support the modelling and code generation of software systems with dynamic evolution requirements, the application of this thesis to PRISMA has extended this model by including dynamic evolution support.

PART II

STATE OF THE ART

CHAPTER III

DYNAMIC SOFTWARE EVOLUTION

3.1 Introduction

This chapter describes the concepts related to dynamic software evolution that are used throughout this thesis. First, section 3.2 presents software evolution, software maintenance and evolutionary processes. Second, section 3.3 introduces dynamic software evolution: the terms that are used in the literature and the different kinds of dynamic evolution, focusing in the granularity and the activeness of changes. Next, section 3.4 presents the main issues related to dynamic evolution: how to preserve the consistence of changes before and after dynamic changes. This involves the safe stopping and the updating of stateful software artefacts. Section 3.5 presents other interesting approaches that address dynamic changes: control systems, autonomic computing and computational reflection. Finally, section 3.6 presents dynamic evolution in the context of dynamic software architectures.

3.2 Software Evolution

The Oxford Dictionary defines **evolution** as *"a gradual process of change and development"*. This definition is very abstract and may have several interpretations. A more general definition, which captures the characteristics of evolution in many situations (including software systems), was proposed by Meir M. Lehman in (Cook et al., 2006):

> *A process of discrete, progressive, change over time in the characteristics, attributes, [or] properties of some material or abstract, natural or artificial, entity or system or of a sequence of these [changes].*

In the context of software engineering, the term *software evolution* has no one widely accepted definition (Mittermeir, 2001). There are two main views on software evolution, referred to as the *what and why* versus the *how* perspectives (Lehman, 1980). The former focuses on the evolution of software as a scientific discipline, and studies the nature of evolution, its impact, and its driving factors. The latter (i.e. the *how*) focuses on the evolution of software as an engineering discipline, and studies the technology, methods and activities that provide means to direct, implement and control software evolution (Mens, 2008). This is the perspective that has been followed in this thesis.

3.2.1 Software Maintenance vs Software Evolution

As a consequence of the perspective of software evolution as an engineering discipline, and the lacking of a standard definition, software evolution is often considered as a synonym of software maintenance (Bennett & Rajlich, 2000), (Chapin et al., 2001), (Mens, 2008). **Software maintenance** is defined in the ISO/IEC 14764 IEEEStd 14764-2006 standard as:

> *The totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage (planning for post-delivery operations, supportability, and logistics) as well as the post-delivery stage (software modification, training, and operating a help desk).*
>
> *(ISO/IEEE, 2006)*

In addition, most of the following evolution-related research themes have been considered as being crucial activities in software maintenance: software comprehension, reverse engineering, testing, impact analysis, cost estimation, software quality, software measurement, process models, configuration management, and re-enginering (Mens, 2008).

However, both terms should not be used as synonyms. From a linguistic point of view, the term maintenance indicates that the software itself is deteriorating, which is not the case. It is changes in the environment or user needs that make it necessary to adapt the software (Mens, 2008). That is, evolution.

From an engineering perspective, software maintenance is generally viewed as a supporting process of a software product, as a way of keeping the product operational and usable. Maintenance activities are to correct faults, to improve performance or other attributes, or to adapt the software product to a modified environment, such as an upgraded operating system. However, the addition of new functionality to a software product is not usually considered as a maintenance activity but an evolutionary activity. For instance, in software product versioning (Apache, 2010), minor version numbers reflect corrections or improvements (i.e. maintenance activities), whereas major version numbers reflect important changes, such as new functionalities (i.e. evolutionary activities).

That is, evolutionary activities are those that involve substantial changes, normally performed at the architectural level, whereas maintenance activities are those that involve minor changes in concrete modules or components. Maintenance activities can be considered as part of the evolutionary activities that are being performed on a software product, but not in the other way round: software evolution has itself a broader view than software maintenance.

3.2.2 Evolution as part of the Development Process

Due to the broader perspective of the term *software evolution*, several authors have preferred to use this term (and not *maintenance*) to refer to a phase of the software life cycle, which lasts from the initial creation of the software product until its eventual retirement or abandonement. This has lead to the following development methods, where evolution is a crucial ingredient: *Evolutionary development* (Gilb, 1981), the *Spiral model* (Boehm, 1988), the *Staged model* (Bennett & Rajlich, 2000), and *Agile Software Development* (Cockburn, 2001), (Martin, 2002). In these models, the phase of software evolution involves the adaptation of the software product to include new user requirements or to perform maintenance activities:

> *[The phase of] software evolution takes place only when the initial development was successful. The goal is to adapt the application to the ever-changing user requirements and operating environment. The evolution stage also corrects the faults in the application and responds to both developer and user learning, where more accurate requirements are based on the past experience with the application.*
>
> *(Bennett & Rajlich, 2000)*

Note the emphasis that is given to the past experience gathered on using the application and how it is considered to generate the new version of the application. As Lehman pointed out: "*evolution is an intrinsic,* feedback driven, *property of software*" (Lehman, 1980). This view is common in all the studies about software evolution, definining it as an iterative, incremental process based on the previous feedback.

Other authors have studied the types of changes that are usually performed to adapt a software product, and have defined what kinds of changes characterize a software evolution activity:

Software evolution occurs when the software maintenance done is of the enhancive, corrective, or reductive types (any of the business rules cluster), or it changes software properties sensible by the customer, i.e., it is of the adaptive or performance types.

(Chapin et al., 2001)

Note that this definition explicitly includes maintenance activities (i.e. enhancive, corrective or performance types) as part of the evolution process. This also confirms the inclusion of the maintenance process as part of an evolution process.

In conclusion, we could define **software evolution** as:

The set of activities and processes that change a previously operational software artefact to correct, improve, extend, or reduce its current functionality to satisfy new requirements based on the past experience with the software artefact and its interactions with the environment.

In this thesis, we prefer to use the term *software artefac*t instead of software product or application to indicate that we can change only a part of a software system. In addition, we have also considered as an influencing factor the past experience gathered on using the software artefact and how it interacts with its environment.

3.3 Dynamic Software Evolution

One of the main difficulties of software evolution is that all artefacts produced and used during the entire software life-cycle are subject to changes, ranging from early requirements over analysis and design documents, to source code and executable code (Mens, 2008). Updating the executable code is the last step to reflect the changes required by the new requirements. In this thesis, the focus will only remain on this last step: how to integrate the changes in the running system.

3.3.1 Changing software artefacts

To understand how a software artefact is evolved or updated, first we need to understand how software systems are built and put into execution. Software systems are constructed from one or more *program modules*, which can be either components, classes or files. The generation of the executable code starts with the *compiling* of modules, a process which generates the binary code of each module. For each module, the compiler includes a header which maps *symbol names* (i.e. variable names or method names) to code fragments. If a module uses/imports another module, then a symbol representing the external module is included in the header, but its mapping is deferred until the linking is performed. The *linking* is the process that takes place after the compilation of the modules: it combines the code of the modules and resolves any reference to externally-defined symbols by matching those references with the appropriate definitions in other modules. When linking finishes, then the code can be executed.

Therefore, to change a software artefact (e.g. a module) that is part of a software system, the following steps must be performed. First the source code of the software artefact must be edited or extended with the required updates and recompiled again. Then, the binary code that has been generated in the compilation process must be relinked to the other modules of the software system, because the references among them have changed. Since the references among modules are defined statically, this linking requires shutting down the software system to integrate the changed modules, that is, to update the references mapped from the old modules to the new modules. This process of change is called **static evolution** or **offline evolution**, because the system needs to be restarted again. This has been the traditional approach for performing the evolution of software (Fabry, 1976), (Buckley et al., 2005). However, this practice has the disadvantage that (i) the state and pending transactions of the running system are lost, and (ii) the entire system will be temporarily unavailable during the restarting process.

3.3.2 Introducing changes at runtime

For some companies, the cost of system shutdown can be prohibitive. Certain safety- and mission-critical systems, such as air traffic control, telephone switching, and high availability public information systems cannot be stopped. Shutting down and restarting such systems for upgrades may incur unacceptable delays, increased cost, and risk (Oreizy et al., 1999). For instance, changing the software that controls a spacecraft or an autonomous robot cannot be done if it means disabling the life-support system or the energy management system. And, disabling a bank-transaction processing system may have significant economic consequences.

In such cases, the support for introducing changes at runtime, without stopping the system, is needed. This was first introduced by R.S. Fabry in 1976, as the need for *"constructing a system in such a way that the programs and the data structures which they manage can be changed without stopping the system"*. This kind of change was originally called *on-the-fly program modification* (Fabry, 1976), (Segal & Frieder, 1993). Since then, other terms have been used in the literature: *dynamic updating* (Segal & Frieder, 1989), *dynamic change* (Kramer &

Magee, 1990), *on-line change* (Gupta et al., 1996), *runtime evolution* (Oreizy et al., 1998), *dynamic evolution* (Malabarba et al., 2000), *live updating* (Vandewoude & Berbers, 2005a), or *online evolution* (Wang et al., 2006).

As it can be observed from these terms, the change on a software system is referred to as *program modification*, *updating*, *change* or *evolution*, whereas the time when the changes are applied are denoted as *on-the-fly*, *dynamic*, *online, runtime,* or *live*. Although these terms are used as synonyms in the literature, there is not a widely accepted definition for the kind of change they refer to.

3.3.3 Dynamic Evolution: Definitions

Generally, most of the existing definitions are very abstract and only refer to the time where changes are applied on a software system:

> *Online software evolution is a kind of software evolution that updates running programs without interruption of their execution.*
>
> *(Wang et al., 2006)*

This definition focuses on the *updating* of a software system at runtime, from the perspective of software maintenance. However, this definition does not describe the nature of such updates, and assumes that the changes can be applied to the running system without any disruption. This is not true, since the subsystems affected by the changes will require to be interrupted somehow to introduce the updates.

Kramer & Magee (Kramer & Magee, 1990) are more explicit and define a *dynamic change* as an evolutionary process, which may involve "*modifications or extensions to a system that were not envisaged at design time*", and which are performed "*without stopping or disturbing the operation of those parts of the system unaffected by the change*". This emphasizes the evolving nature of software systems and the minimal disruption on the execution of the other elements of a system.

A different focus is taken by Gupta, Jalote and Barua, who defined *on-line change* as an instantaneous rather than an evolutionary process:

> *An on-line change from program* ∏ *to* ∏' *at time* t *using the state mapping* S, *in a process* P *(executing* ∏*) is equivalent to the following sequence of steps:*
>
> *1)* P *is stopped at time* t *in state* s;

> *2) The code of* P *(which, until now, was the program* ∏*) is replaced by the program* ∏'*, its state is mapped by* S *and* P *is then continued (from state* S(s) *and with the code of* ∏'*).*
>
> *(Gupta et al., 1996)*

This definition focuses on the transference of the state among the different program versions. The change is viewed as a replacement or updating operation, where a new program (or module) replaces another, and the previous state is mapped to the data structures of the new module. This is the perspective taken in the so-called *dynamic updating* and *live updating* approaches. However, this definition considers an update operation as a change that modifies the whole program, so it requires stopping the entire system. This is not acceptable in medium-sized and large-sized software systems: only the affected parts or structures should be stopped.

In this thesis, we propose the following definition for dynamic change:

> **Dynamic evolution** *is a process of gradual change that is performed on a previously operational software system to correct, improve, extend or reduce part of its functionality, which occurs during its execution, without disturbing those parts of the system unaffected by the change, and which preserves the system's integrity.*

This definition explicitly indicates that:

- The changes have an evolutive, incremental nature.
- The system to evolve was previously operational. Otherwise, if the system was not operational (e.g. it may be running but being hanged or crashed), then static evolution should be used (i.e. changes can be performed at design-time).
- Changes may involve update operations (i.e. corrections and improvements), addition of new functionalities, or removal of previous functionalities.
- The change process is performed while the system is running, without the need of stopping it.
- Only those parts of the system which are subject to changes should be stopped. The other parts, which are running, should not be disrupted by the change process.

- System integrity must be preserved: the system should continue working normally, and be left in a consistent state. When updating a part of the system, its previous state should be preserved or migrated whenever possible.

We have preferred to use the term *dynamic evolution* instead of the other terms proposed in the literature (e.g. on-line change, dynamic updating, dynamic change, runtime evolution, etc.), because it describes better the change process. On the one hand, the term *evolution* refers to progressive improvement, as a consequence of natural selection mechanisms. Software products evolve, or change progressively, while they are still alive, i.e. being used. This is not reflected by the other terms (i.e. modification, change, and updating), which suggest the realization of instantaneous, occasional changes, instead of continuous, progressive improvements. On the other hand, the term *dynamic* refers to objects that are in motion, in activity or in progress. This perfectly defines the nature of a runtime change: changes that are performed while the software system is running, engaged in some activity, and without stopping it.

3.3.4 Kinds of Dynamic Evolution

As described in the previous section, dynamic evolution is a term that is used to describe those software changes or improvements that are done at runtime, without stopping the software system. It is a general term that is not constrained to changes on a specific kind of software artefact or to a specific granularity of change.

There are other terms in the literature that denote specialized types of dynamic change, and which should not be incorrectly used as synonyms for dynamic evolution, which has a more generic meaning. One of these terms is **dynamic adaptation**. For instance, Keeney uses this term as "*the act of changing the behaviour of some part of a software system as it executes, without stopping or restarting it*" (Keeney, 2004). Walter Cazzola et al. define dynamic adaptation as the ability of a software system "*to adapt itself to environmental changes by adding new and/or modifying existing functionalities, avoiding a long out-of-service period for maintenance*" (Cazzola et al., 2004). These definitions have some points in common with the definition of dynamic evolution previously presented. However, the term *dynamic adaptation* is a specialized kind of dynamic change that is performed by means of adaptors (Canal et al., 2006). An adaptor is a specific computational entity developed for guaranteeing that a set of mismatching components will correctly interact (Canal et al., 2008). Thus, dynamic adaptation is defined as *the process of changing a software system at runtime in a nonintrusive way, by means of adaptors, and without modifying the* *code of components*. By contrast, dynamic evolution may be performed either intrusively, by modifying the code of components (e.g. to add new functionality), or non-intrusively, by means of adaptors.

Next, other specialized terms are introduced, which are subclassified by the granularity of changes (*dynamic reconfiguration*, *dynamic type evolution* and *dynamic updating*), and by the activeness of such changes (*reactive/ad-hoc evolution*, *programmed proactive evolution*, *non-programmed proactive evolution*). Additional attributes for characterizing dynamic changes can be found in (Buckley et al., 2005) and (Andersson et al., 2009a).

3.3.4.1 Granularity of changes

Dynamic evolution can be performed at different granularity levels. Granularity refers to the scale of the artefacts to be changed and can range from very coarse, through medium, to a very fine degree of granularity (Buckley et al., 2005):

- **Coarse granularity changes** are those changes that are performed at the level of system architecture. That is, changes that may impact several subsystems, such as modifying an entire subsystem or adding a new functionality. These kinds of changes are performed by *Dynamic Reconfiguration*, which addresses the reconfiguration of a software architecture at runtime, by adding/removing components, connectors and their interconnections.
- **Medium granularity changes** are those changes that impact the composition of components, modules, or classes, and all of its instantiations. These changes are addressed by the *Dynamic Evolution of Types*, which provides support to the modification of types (i.e. specifications) at runtime and their instances.
- **Fine granularity changes** are those changes that are performed at the level of variables, statements or methods. These changes are transversal to the other granularity levels and are implicitly provided by the upper levels. They are generally addressed by *Dynamic Updating* approaches (Hicks & Nettles, 2005).

Generally, existing approaches only provide change management at a single granularity level. However, these levels complement each other and should be combined to provide critical software systems with the maximum degree of flexibility.

3.3.4.2 Activeness of change

Dynamic changes can be driven in two different ways: *reactively* or *proactively* (Buckley et al., 2005).

On the one hand, **reactive changes** are those changes that are driven by an external agent (usually the developer), typically by means of a user interface or an external tool. Support for reactive changes is recommended to be able to introduce unforeseen changes, i.e. changes not initially predicted during the design of a system. In the area of software architecture, this kind of change is also referred as *ad-hoc reconfiguration* (Endler & Wei, 1992). Most of the works addressing dynamic updating support this kind of change (Segal & Frieder, 1993).

On the other hand, **proactive changes** are those changes that are driven autonomously by the system when some specific conditions or events apply. An example of the use of such changes is to provide system dependability: if a component instance does not adequately respond, the system might change its connections to another suitable component instance, or recreate the component instance again. Two subtypes of proactive changes can be distinguished, depending on whether these changes are previously programmed or not:

- **Programmed evolution**. Changes are defined at design-time, and are activated when a certain condition or event applies. In software architecture, this kind of change is referred as *programmed reconfiguration* (Endler & Wei, 1992), and are described by means of reconfiguration specifications. A reconfiguration specification describes *when* the architecture of a system should change (e.g. in response to a certain event or a state change), and *what* kind of changes must be performed on this architecture for each situation. Most of the works addressing dynamic reconfiguration support this kind of change (Bradbury et al., 2004).
- **Non-programmed/Generative evolution**. Changes are automatically synthesized at run-time by the system, which also decides when to load and use them. This is the most powerful kind of change a system may exhibit, but also the most difficult to control. Although this does not imply the presence of intelligence, a system provided with this kind of evolution will exhibit intelligent behaviour (Pollack, 2006) because of its ability to synthesize new behaviours in response to either internal or external stimuli. This kind of evolution is still very challenging. For this reason, only a few works provide some kind of support to this kind of evolution. An example is the work of Sykes et

al. (Sykes et al., 2008), which provides support to automatically synthesize reconfiguration specifications according to high-level goals.

There are already some techniques that could be used to address non-programmed evolution. For instance, the generation of behaviours from observed internal or external facts can be addressed by means of *Inductive Logic Programming*: a rule learning technique based on logical programming that uses previous background knowledge and existing facts or observations to infer (i.e. generate) new rules (i.e. behaviours) at runtime (Bergadano & Gunetti, 1995). Other kind of promising techniques are those based on *evolutionary computation*, which are techniques inspired on darwininian evolution processes for automated problem solving. In this context, the individuals of the population are program specifications which are evaluated and evolved iteratively until a user-defined task is achieved. Some works have already exploited these techniques for generating some software artefacts. For instance, (Wong & Mun, 2005) have analysed the use of genetic programming to produce (logical) recursive programs, whereas (Weise et al., 2009) have used genetic programming to generate UML models of programs. Likewise, (McKinley et al., 2008) uses digital evolution techniques to generate programs.

Reactive and proactive changes are complementary: both should be supported to allow a system to introduce unforeseen changes or updates (i.e. reactive change support), and to reconfigure itself autonomously in response to certain situations (i.e. programmed proactive change support). The combination of both kind of activeness is beneficial for complete dynamic evolution support, since it would allow us to introduce at runtime new component types and reconfiguration specifications (i.e. proactive change specifications) that were not envisaged at design-time.

3.4 Main Issues of Dynamic Evolution

When addressing dynamic evolution of software systems, different issues must be addressed. The two most important issues which must be dealt with are reaching a consistent application state, and the transferring of state. On the one hand, **application consistency** refers to the safe stopping of the running software artefacts, so that they can be changed with minimal disruption in the normal operation of the system. On the other hand, the **transferring of state** refers to the migration, or the transformation, of the internal structure and information content of a software artefact at runtime. Next, these issues are presented in detail.

3.4.1 Safe Stopping of Running Systems

One of the main issues that must be faced when addressing dynamic evolution is to place a system in a consistent state before and after dynamic change. The removal or replacement of a system element at runtime (e.g. a component, a type, or a relationship) may cause that some service requests that were pending to execute could be lost (if they were to be executed on a removed element), or incorrectly executed (if they were to be executed on an element which has been changed). This will lead the state of a system to be inconsistent, and probably, to evolve towards a failure.

For this reason, before performing dynamic changes on a running system, the affected elements must be placed in a (deactivated) *safe state*, which must guarantee that changes on these elements will not introduce inconsistencies in the system. This process must be performed with the minimum disruption on the elements that are unaffected by the changes. The conditions that an element (either stateless or stateful) must fulfil to be in a safe state are provided by the so called *safe stopping criteria*. This section presents the most extended safe stopping criteria, quiescence and tranquillity, and introduces other approaches that have been proposed.

3.4.1.1 Quiescence

One of the most extended and influential safe stopping criterion is *Quiescence*, proposed by Jeff Kramer and Jeff Magee (Kramer & Magee, 1990). Their main contribution is the introduction of a number of requirements to ensure consistency when reconfiguring a distributed system. In their model, a system is seen as a directed graph whose *nodes* are system entities and whose *arcs* are connections between those entities. Nodes are processing entities that can initiate and/or service *transactions*, which consist of a sequence of messages that must be executed atomically (i.e. all messages are executed, or none of them).

There are two kinds of transactions: independent and dependent. A transaction is *independent* if its completion does not depend on any other (possibly nested) transactions with other nodes. A transaction is *dependent* if its completion may depend on the completion of other consequent transactions. Cycles are not forbidden, but it is assumed that transactions complete in bounded time and that deadlocks are avoided.

To perform a dynamic change, all the nodes that are going to be affected by such change must reach local consistency first. Local consistency in a node will be achieved if it has no partially completed transactions. And, to be able to determine if there are partially completed transactions, it is assumed that the initiator of a dependent transaction is informed of the completion of its consequent transactions.

Kramer and Magge abstract the status[5] of an application into a set of different configuration statuses for each node:

- **Active status**. A node in this status can *initiate*, *accept*, and *service* transactions. This is the normal behaviour.
- **(Generalized[6]) Passive status**. A node in this status must continue to accept and service transactions and *initiate consequent transactions*, but:
    - It is not currently engaged in a (nonconsequent) transaction that it initiated, and
    - It will not initiate new (nonconsequent) transactions

  That is, a passive node: (i) accepts and services pending transactions from connected nodes to allow them to complete outstanding transactions; (ii) has finished the nonconsequent transactions it initiated; and (iii) only initiates transactions that are required by other pending transactions (i.e. it only initiates nested transactions).
- **Quiescent status**. A node will be in this status if:
    - It has passive properties (i.e. it is not engaged in a transaction that it initiated and will not initiate new ones)
    - It is not currently engaged in servicing a transaction,
    - No transactions have been or will be initiated by other nodes that require service from this node.

  In this status a node is both *consistent* and *frozen*. It is consistent because the node does not contain the results of partially completed transactions, and is frozen because the node state will not change as a result of new transactions.

To move a node Q from the active status to the quiescent status, the following nodes must be directed towards the passive status first:

(i) the node Q;

[5] As introduced by (Vandewoude et al., 2007), it will be used the distinction between the internal *state* of an element (which is migrated or transformed in the evolution process) and the *status* that describes its condition with respect to the evolution process.

[6] The most extended definition of passive status is restricted to only independent transactions (i.e. which cannot be nested). For the sake of generality, the generalized definition is provided, because it takes into account dependent transactions.

(ii) all nodes which can directly initiate transactions on Q, i.e. all nodes with connection arcs directed towards Q;

(iii) all nodes which can initiate dependent transactions which result in consequent transactions on Q.

This is called the *enlarged passive set* EPS of a node Q, denoted EPS(Q). Kramer and Magee demonstrated that, in a system with nested transactions and assuming that these transactions complete in bounded time, a node Q can move towards the quiescent status in bounded time if all the nodes in EPS(Q) are passivated.

Kramer and Magee have shown that the quiescence requirement is sufficient to ensure consistency, and that quiescence is reachable in finite time. The model was implemented and tested in the Conic environment for distributed programming (Magee et al., 1989). It has been the basis of many other systems, such as (Bidan et al., 1998) and (Moazami-Goudarzi, 1999).

However, the main disadvantage of the model of Kramer & Magee is that enforcing quiescence in a system with nested transactions often causes serious disruption to the running system (Vandewoude et al., 2007). Not only must the node that is to be updated be put in a passive status, but this is also the case for every node that is directly or indirectly capable of initiating transactions on this node. This results in a large number of nodes that need to be passivated.

Another disadvantage is that is not always feasible to assume that a node will have knowledge of whether its actions are part of a transaction initiated by another node, that is, to identify those transactions that are consequent and those that are not.

3.4.1.2 Tranquillity

The concept of Tranquillity was proposed by Yves Vandewoude, Peter Ebraert and Yolande Berbers as a solution to reduce the constraints that the quiescence condition exhibits (Vandewoude et al., 2007). The tranquillity condition is easier to obtain, less disruptive than quiescence, and still sufficient to ensure consistency before changes.

Tranquillity is based on the following observation. A node that participates in an active transaction can be safely replaced if: (i) it is certain that it will not further participate in the transaction, and (ii) it has not yet participated in the transaction. It is assumed that, in a replacement operation, transactions that have not yet begun may be executed by the new version. In these cases, it is not needed to wait until the active transaction finishes, and the replacement could be performed earlier.

- **Tranquillity status**. A node will be in this status if:
    - It has passive properties (i.e. it is not engaged in a transaction that it initiated and will not initiate new ones),
    - It is not actively processing a request,
    - None of its adjacent nodes are engaged in a transaction in which it has *both already participated and might still participate in the future.*

Tranquillity is a weaker condition than quiescence: it relaxes some of the constraints quiescence defines in order to reduce the number of nodes that need to be passivated. For this reason, it is said that quiescence implies tranquillity but not vice versa. Tranquillity does not imply quiescence because it does not forbid adjacent nodes to initiate new transactions that involve the *tranquile* node. Adjacent nodes are only required to finish those transactions on which the node that is being stopped has participated and might participate in the future. In other words, tranquillity does not require to completely passivating all the adjacent nodes.

However, one of the main disadvantages of the tranquillity condition is that it is only valid for a subset of change operations: addition, linking and replacement of nodes. Tranquillity does not guarantee consistency when deleting or unlinking nodes. The reason is the following: a node in the tranquillity status may be engaged in an ongoing transaction initiated by one of its adjacent nodes, but on which it has not participated yet. If the node is unlinked or removed from the system, then the ongoing transaction will fail when trying to request a service from this node, leaving the system in an inconsistent state.

Another disadvantage is that dependent transactions are not well addressed. A node in a tranquil status is not executing code and can be only involved in an ongoing transaction if its participation in this transaction is: (i) finished, (ii) not yet begun, and (iii) *part of a subtransaction* (Vandewoude et al., 2007). A subtransaction, or consequent transaction, is a transaction that is initiated by a participant of a transaction as part of its response to a message that it process from the original transaction. The authors assume that subtransactions are independent of its original transaction and may be executed by a different version than the original transaction. This is not always true: a subtransaction may use information that has been calculated by the original transaction, so the subtransaction is *not independent of its predecessor.* For instance, suppose that a node X, at time t_i has finished its participation in an ongoing transaction, and as a result of this participation, the internal state of X has the value v. Then, at time t_{i+1} another node Y, which is engaged in the active transaction, starts a subtransaction which involves *X* and changes its state to *v'*. According to the tranquillity condition, the node *X* could be replaced at time t_i by a new version. However, if the new version does not migrate correctly the previous state *v*, an inconsistency with *Y* may be produced. This is still an open issue.

Finally, another disadvantage is that it is not guaranteed that a tranquil status will ever be reached. This is the case when a node is used in an infinite sequence of interleaving transactions (Vandewoude et al., 2007). In addition, tranquillity is not stable: as soon as a node achieves a tranquil status, all interactions between that node and its environment must be blocked. By contrast, quiescence is stable: once a node is quiescent, it must be explicitly reactivated. This is because tranquillity can occur naturally, whereas quiescence must be actively driven.

For all these drawbacks, tranquillity should be considered as a complementary criterion to quiescence. Tranquillity can be reached quickly and with less disruption on a system than quiescence. However, in these cases where tranquillity is not feasible, a fallback mechanism to quiescence should be provided. An example of the implementation of tranquillity and quiescence together is provided by DRACO (Vandewoude et al., 2003), (Vandewoude et al., 2007), (Vandewoude, 2007). DRACO is an extensible and modular component-based framework which provides a reusable module, the *Live Update Module*, which supports tranquillity and the fallback to quiescence if tranquillity is not reached in finite time.

3.4.1.3 Other approaches for Safe Stopping

Based on the work of Kramer and Magee, Moazami-Goudarzi (Moazami-Goudarzi, 1999) presents an alternative to reach a safe state with significantly less disruption to the running system. However, he assumes that components never interleave transactions (i.e. a component never participates in a new transaction while another transaction is still in progress). Under this assumption, a quiescent state can be reached much more easily by simply blocking messages when no transactions are being serviced. A similar approach is used in object-oriented contexts: the dynamic replacement of a class is delayed until no methods of this class are active (Andersson et al., 1998), or if there are methods that are active, they remain unchanged in the new version of the class (Malabarba et al., 2000).

The principle of a safe state has been also used in systems that support dynamic reconfiguration on top of CORBA. For instance, (Bidan et al., 1998) consider node consistency as integrity of RPC (i.e. Remote Procedure Calls): all RPC's initiated by nodes affected by a reconfiguration must be completed before the changes are performed. They consider the passivation of links instead of the passivation of sets of nodes. The advantage is that multi-threaded components can continue to function, since only the threads that use a passivated link are required to block. The disadvantage is that all the RPC requests must be independent, that is, no nested RPC's are allowed and the reconfiguration of systems with re-entrant invocations is not supported.

Another approach based on CORBA is the work of (Almeida et al., 2001). They consider that a system is in a safe state when each node that will be affected by changes is not currently involved in interactions and will not be involved in interactions. They distinguish among between two types of nodes: purely reactive or active. Reactive nodes are only allowed to initiate requests that are causally related to incoming requests. Active nodes may proactively initiate new requests. However, active nodes must implement a functionality to switch to a reactive modus. Then, requests to affected nodes are selectively queued by only allowing re-entrant requests to be delivered to the node.

Other approaches have taken an invasive approach: the developer must define in advance locations in the code where changes may take place (Hicks & Nettles, 2005). In PODUS (Segal & Frieder, 1993), dynamic change is applied at procedure level. Only procedures which are both syntactically and semantically inactive can be updated. A procedure is syntactically inactive if it is not on the runtime stack and its new version does not call a procedure that is on the runtime stack. A procedure is semantically inactive if it does not depend on the tasks that are performed by a procedure that is on the runtime stack. Since these dependencies cannot be derived from the code, a list with semantic dependencies must be specified by the programmer before the update takes place. That is, the programmer must specify which procedures must be updated concurrently.

A similar approach has been used in the OpenCOM component model (Coulson et al., 2004), (Coulson et al., 2008) to deal with the dependencies that multithreaded components may introduce in a system. Pissias and Coulson (Pissias & Coulson, 2008) have addressed the implementation of the quiescence criterion in the OpenCOM component model without introducing architectural restrictions. Their design is based on interception in connectors, and makes use of meta-data and reflection services to obtain information about the nodes that are involved in an ongoing transaction. However, the developer is required to tag those services that may be blocking or unblocking in order to prioritize those calls that will enable the finalization of pending requests.

Gomaa and Hussein (Gomaa & Hussein, 2004) introduced a set of design patterns for dynamically reconfigurable systems, most of which are based on the concept of quiescence presented in this section. The contribution of their work is that they specify, by means of UML state diagram templates, the behaviour required to reconfigure different architectural styles: master/slave, server/client, centralized and decentralized architectures.

3.4.2 Updating Stateful Artefacts

Another important issue that must be faced when addressing dynamic evolution is the update of *stateful*[7] artefacts. In case of stateless elements, the updating is simple: once their interactions have been safely stopped (i.e. the affected elements have been quiesced), they can be replaced with the new version and reactivated again. However, in case of stateful elements, the updating requires their previous state to be *transferred* to the new version. And this is not easy: the issue is to identify which is the relevant information of the previous version, how to obtain it, and how to migrate it to the new data structures of the new version. This is known as the problem of *state transfer*.

State transfer involves extracting the runtime state of an active element and subsequently using this information to initialize the new version. The state that needs to be transferred between versions depends on the actual state of the application when the update takes place. If the update takes place in the middle of a method execution, the runtime stack, CPU registers and exact location in the method need to be preserved. If no methods are active, only persistent state needs to be copied (global variables, instance variables in object oriented languages, etc.). An important reason to drive an application to a *safe state* (see previous section) is that it minimizes the amount of control state that needs to be transferred.

However, the challenging issue is not with *copying* the data among the old version and the new version, but with *translating* the data among the different data structures. If a data structure has significantly changed between two versions, a state or data transformation process will be necessary. The problem is that this process is application specific: it requires an understanding of the semantics and meaning of the data being processed. The challenge has been in finding a general purpose algorithm to deal with such state translation.

[7] The term “stateful" (Vandewoude, 2007), (Hammer, 2009) usually refers to the existence of internal state, which cannot be directly observed from outside, and influences the communication behaviour of the element. This internal state may be changed as a consequence of the interactions with other elements, or spontaneously due to proactive behaviour.

Three alternatives have been proposed: no state transfer, delegated state transfer, and automated state transfer. These alternatives are presented in the following subsections.

3.4.2.1 No State Transfer

Given the difficulty that entails the state transfer problem, some approaches have proposed to allow old code and new code to be intermixed (Segal & Frieder, 1993), (Andersson et al., 1998), (Bierman et al., 2003), and the choice of which to execute to be determined automatically. Thus, existing instances are not transformed, and are used exclusively by older versions of the code. Newly created instances are always of the most recent version, and are handled by the new code. This technique is called *Passive Partitioning* (Gray & Hjálmtýsson, 1998), (Malabarba et al., 2000). It is often combined with type renaming, to allow multiple versions to coexist. This technique is well suited for distributed systems where it is not always feasible to locate all existing instances of an old version.

The advantage is that is efficient to implement and to adopt, and sufficient for some cases. The disadvantage is that the updates are deferred until new instances are required. Instances using old versions do not benefit from the updates introduced by new versions. In addition, this semantics introduces the difficulty of dealing with orderly transitions among versions. It becomes quite difficult to do so given the possibility of many versions of code and state interacting together (Hicks & Nettles, 2005).

3.4.2.2 Delegated State Transfer

Another solution is to take a hybrid approach: automatize the migration of old instances to the new version, but delegate the complexity of the transformation of data structures to the designer of the update. For instance, the work of Gupta and Jalote (Gupta & Jalote, 1993) completely implements automated state transfer for procedural systems (i.e. the copy of the data). However, their approach requires that the state structure of the new version remains unchanged, i.e. state transferring is supported, but not state transformation.

If the state structure of the new version changes, the designer of the update is required to implement a *transformation function* to convert the old state structure to the new version. Generally, this transformation function does the following steps:

(i) is provided with the old instance to transform,
(ii) gets access to the internal state of this instance (which internal structure is known by the developer), and

(iii) maps the attributes of the old instance to an instance of the new version.

Some approaches provide facilities to help the developer in this task. For instance, in JDRUMS (Ritzau & Andersson, 2000) a conversion skeleton class is generated which includes the attributes of the old and the new type version; the developer then defines manually the mappings among the attributes. A similar approach is followed in (Hicks & Nettles, 2005), by automatically generating the method template that must be filled in.

The migration of old instances may be fully automated, i.e. all the old instances are converted, or selectively defined, i.e. the developer decides which instances are converted and which not. The former is called *Global Update* and the latter *Active Partitioning* (Malabarba et al., 2000). In *Global Update* approaches, all existing instances are migrated, so it prevents the existence of multiple versions of a given element to co-exist at any given time. An example of such an approach is the work of Malabarba et al. (Malabarba et al., 2000). In *Active Partitioning* approaches, the developer specifies which instances will be converted and when/how this conversion must take place, leaving the others in the previous version. An example of such kind of approach is the work of (Gray & Hjálmtýsson, 1998), which does not convert instances by default, but the programmer is given the option to explicitly convert instances himself.

3.4.2.3 Automated State Transfer

Finally, another solution is to fully automate the state transfer process: the dynamic updating system is capable of extracting the state information of old versions and automatically translating this information to the data structures of the new version.

However, such complete automation is not always feasible. On the one hand, the state of an element has always a semantic meaning, which is determined by the interpretation of the developer and cannot be automatically derived from the source code. This semantic relation should be interactively provided to be able to transfer the state between two versions. On the other hand, the new data structures may contain information that is not available in the old data. For instance, it could be imagined the addition of a time stamp field to a data structure to indicate when it was created. Dynamically transforming existing instances to add this new field will be impossible because the information simply does not exist at dynamic-update time. Bloom and Day (Bloom & Day, 1993) have explored some of the theoretical limitations of state transformation, whereas Adamek and Plasil (Adamek & Plasil, 2005) have explored how to formally verify at runtime the atomicity of dynamic updates.

Nevertheless, some approaches exist that provide a semi-automatic process for the state transfer problem. One of these approaches is DeepCompare, a tool developed by Yves Vandewoude and Yolande Berbers (Vandewoude & Berbers, 2005), (Vandewoude, 2007). Their approach consists of a (pseudo-interactive) static analysis based on heuristics which takes place after the design and implementation of a new component version. The analysis automatically detects similarities in the source code of the old version and the new version with minimal user effort, and maps corresponding structures between them. Then, this information is embedded in the binary of the new component version and used at runtime by a fully automatic and generic state transfer algorithm. Experimental validation showed that their tool is capable of automatically matching the corresponding state structures between different versions in the 95% of cases, reducing the time required to implement the state transfer logic. The only disadvantage is that this process is performed at design-time, and the access to the source code of the old version is required.

The approach proposed by Vandewoude & Berbers is based in a *Direct State Transfer* (Vandewoude & Berbers, 2005): the source code of the old version is used directly for extracting, interpreting and converting the information contained in the version to be replaced. Another feasible approach is the use of *Indirect State Transfer*: old version instances provide a service to export its state in an abstract representation, which is platform independent and generally easier to analyse. The advantage is that the source code of the old version is not needed, so encapsulation is preserved. This approach could benefit from the findings in the area of digital libraries and the semantic web, since they address a similar problem to state mapping: the automatic generation of mappings among ontologies. For instance, the work of (Llavador & Canos, 2007) presents an approach to automatically generate XSL transformation templates using semantic relations among schemas.

Other relevant works that could be applied to this context are those addressing the evolution of schemas in relational and object-based databases, such as the works of (Staudt, 2000) or (Boronat et al., 2004). For additional information of which component-based frameworks provide state-transfer support and a comparison among them, the reader can refer to the work of M. Hammer (Hammer, 2009).

3.5 Other approaches for dynamic change management

This section introduces the key findings from other research fields that have been relevant for this work: *Control Systems*, *Autonomic Computing*, and *Computational Reflection*. These fields are relevant because they have dealt with the process of changing running systems: the adaptation of physical devices (Control Systems), the self-management of Information Technology infrastructures (Autonomic Computing), and the development of highly dynamic applications (Computational Reflection). These fields have in common the presence of adaptation loops to supervise the running system and adapt it accordingly to the system ouputs or some external stimuli.

3.5.1 Control Systems

Control systems are used for building physical devices which need to adapt their output to some degree of variable input. An example is the control system that manages the movement of the wheels of a mobile robot. The motors of the wheels turn at a variable speed; the control system increases this speed when the target is far, and decreases when the robot is approaching to its objective.

A *control system* is defined as a process which supervises the execution of another process and adapts it according to received stimuli. This adaptation is generally performed by means of the configuration of different parameters of the managed process.

Depending on the source of the stimuli, a control system can be *open* or *feed forward*, if the stimuli are received from the environment; or *closed* or *feedback*, if the stimuli are received from the outputs of the controlled process (Brosilow & Joseph, 2002). Both kinds of control systems can be combined for building devices that get the information both from their environment and from their outputs (see Figure 3.1).

In this case, the control system is composed of two controllers: a feedback controller, which processes the outputs of the process, and a feedforward controller, which processes the signals from the environment. The decisions of both controllers are weight according to the goals of the built device.

The interesting point here, as stated by other authors (Hellerstein et al., 2004), (Morrison et al., 2007), is that these ideas are also applicable in the development of adaptive/reconfigurable software systems. Software systems have the benefit of being more malleable than hardware systems. Applying these (elementary) notions of control systems for modelling the dynamic evolution of software architectures we would obtain that: the controlled process would be the software architecture, and the controllers would be the mechanisms capable of dynamically changing this architecture. Thus, a feedforward evolution controller would be an element that uses the external stimula to integrate unforeseen changes in the architecture (at runtime). And a feedback evolution controller would be an element that uses the internal stimula to optimize the execution of the architecture and react to foreseen changes. These ideas have been taken into account to develop the proposal presented in this thesis.

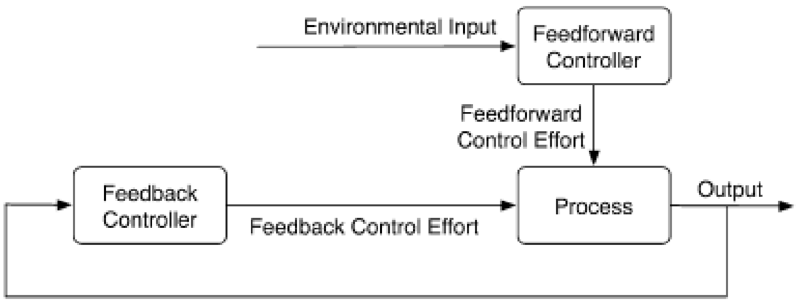


Figure 3.1. A Feedback and Feedforward control system (Morrison et al., 2007)

3.5.2 Autonomic Computing

The concept of *Autonomic Computing* was created from the need of releasing Information Technology administrators from the burden of performing repetitive management & configuration tasks. This concept emerged as a result of an IBM's research project (IBM, 2001), and was defined by Jeffrey O. Kephart and Dave Chess as:

> *Computing systems that can manage themselves given high-level objectives from administrators. [...] The term autonomic computing is emblematic of a vast and somewhat tangled hierarchy of natural self-governing systems, many of which consist of myriad interacting, self-governing components that in turn comprise large numbers of interacting, autonomous, self-governing components at the next level down.*
>
> *(Kephart & Chess, 2003)*

Therefore, the goal is to build hardware/software systems whose resources (files, databases, disks, etc.) are automatically managed, thus reducing their dependence on humans to constantly monitorize and configure such resources. These automatically managed systems would be able to monitorize their operation and to adjust themselves to guarantee a certain quality of service.

Kephart and Chess (Kephart & Chess, 2003) stated that an autonomic system is characterised by the following four facets of self-management:

- *Self-Configuration*. The components that are deployed in a system are capable of installing and configuring themselves, following user-defined high-level policies. They register their capabilities in the system so that other components can either use it or modify their own behavior appropriately. The rest of system components adjust seamlessly themselves to integrate or use the capabilities provided by new components.
- *Self-Optimization*. Components periodically monitorize their behaviour looking for potential performance improvements, by means of the variation of their configuration parameters, or looking for the latest updates.
- *Self-Healing*. A system is able to detect, diagnose and repair localized software and hardware problems, by using knowledge about the system configuration or log information. Repairs can be performed by applying available patches or the automatic reinstallation of affected modules.
- *Self-Protection*. A system protects itself from malicious attacks or cascading failures. It uses early warnings to anticipate and prevent global failures.

Other authors have distinguished many other facets for self-management, which have been collectively named as self-*systems (Babaoglu et al., 2005). Since the full categorization of the wide range of facets proposed in the literature is so difficult (Lin et al, 2005), (Cuesta & Romay, 2010), several authors have even suggested to refer only to generic dimensions (Andersson et al, 2009a).

The realization of these facets (also known as self-management attributes) are generally achieved by means of the definition and composition of *autonomic managers*, which are the main building blocks of an autonomic system. An autonomic manager is defined as:

> *[A software artefact that] automates some management function and externalizes this function according to the behaviour defined by its management interfaces.*
>
> *(IBM, 2006)*

That is, an autonomic manager is an element that automatizes the management of a resource, according to a set of high-level policies defined through its management interfaces. This resource can be either a non-autonomic resource (a file, a database, an application, etc.) or another autonomic element. This allows the compositionality of autonomic managers, i.e. autonomic managers that manage other autonomic managers.

An autonomic manager implements an intelligent control loop, which consists of the following functions (see Figure 3.2):

- *Monitor*: provides mechanisms that collect, aggregate, filter and report details (such as metrics and topologies) collected from the managed resource. This is achieved by means of multiple sensors provided by the resource, which can include aspects of hardware instrumentation (e.g. temperatures at various points within the hardware platform), ambient information (e.g. environmental temperature, physical intrusion of the device), or software components (e.g. various performance-monitoring counters of the operating system, or specific counters for application monitoring). The data can be acquired from sensors by polling at specified intervals or can be collected asynchronously when specified thresholds are exceeded.

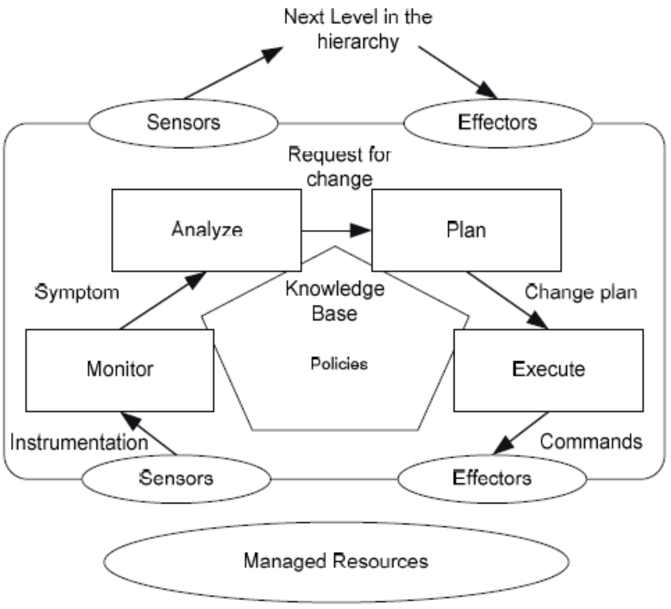


Figure 3.2. Internal Structure of an Autonomic Manager (Tewari & Milenkovic, 2006)

- *Analyze*: contains the intelligence required to interpret and correlate the information provided by the Monitoring data. This allows the autonomic manager to learn about the IT environment and help predict future situations. It may use historical data and compare with current state to detect significant changes, in order to perform performance adjustment or work around anticipated faults.

- *Plan*: provides the mechanisms that construct the set of control actions needed to achieve goals and objectives, by using policy information to guide its work.
- *Execute*: this functionality receives the series of action steps from the planning element, and puts the plan into action. It activates the appropriate control points, or effectors, on the managed resource following the proper sequence and timing.
- *Knowledge base*: This serves as a repository of knowledge, such as historical data and policies, which can be utilized by the other elements in their operation. Knowledge can be obtained from multiple sources: from other autonomic managers, a management console and operator, or can be obtained using machine-learning techniques from prior observations of system states and corrective changes that were found to be effective.

As can be observed from the figure, an autonomic manager has sensors and effectors wich act on the managed resource (see the bottom of Figure 3.2). In addition, an autonomic manager also provides sensors and effectors (see the top of Figure 3.2) to allow its management and configuration by other autonomic managers. This allows the hierarchical composition of autonomic managers in order to achieve the global facets of self-configuration, self-optimization, self-healing and self-protection by the orchestration of several autonomic managers.

A good illustrating example of a possible composition of autonomic managers to build an autonomic system is depicted in Figure 3.3. This figure shows a layered architecture composed of: (i) the resources (servers, storage, network, databases, etc.); (ii) management interfaces (i.e. sensors and effectors) which act on the resources; (iii) single-unit autonomic managers, i.e. those that manage a non-autonomic resource; (iv) multiple-unit autonomic managers, which manage several single-unit autonomic managers to achieve one self-* facet, or which solves conflict among different self-* facets; (v) the system administrator, that provides the high-level policies; and (vi) the knowledge sources.

The area of autonomic computing is still an active area of research: several challenges remain open and no a commonly accepted framework exists yet. A good introduction to the key findings are the survey of (Huebscher & McCann, 2008), the introduction of (Muller et al., 2008), or the standards that could be used to implement such elements (Tewari & Milenkovic, 2006).

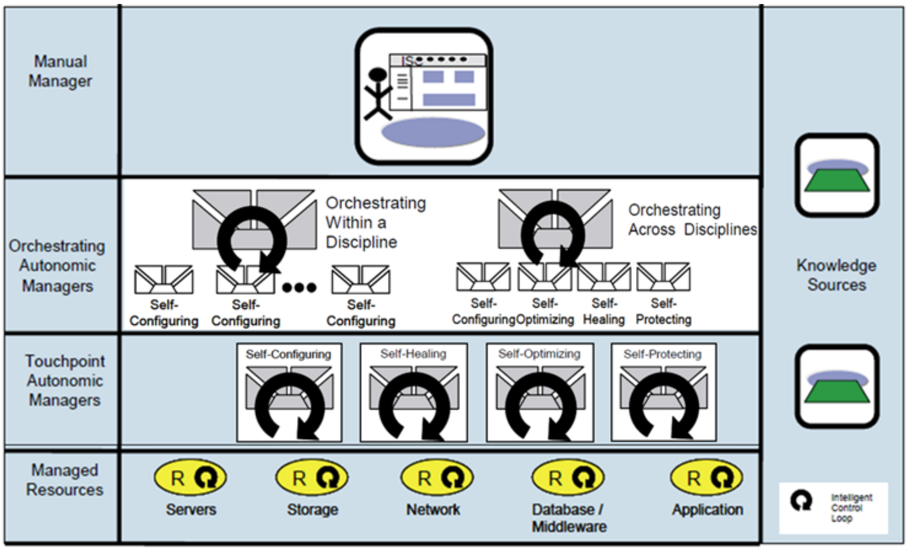


Figure 3.3. Reference architecture for Autonomic Computing (IBM, 2006)

3.5.3 Computational Reflection

Reflection is a concept arised from the artificial intelligence field, as the ability of a system *to reason about and act upon itself*. Reflection is about meta-computation, i.e. computation about computation. This was considered as an emergent property responsible for intelligent behaviour. The foundations were originally laid out by Brian C. Smith in the 80's (Smith, 1982). He proposed the use of reflection in the context of programming languages and developed the language *Lisp-3*, which quickly became famous in the functional community (Demers & Malenfant, 1995).

Reflection become popular and spread to other fields, such as distributed systems, operating systems, and middleware (Kon et al., 2002). This was mainly thanks to the contributions of Pattie Maes, who contributed to summarize the existing notions about reflection:

> *Computational Reflection is the activity performed by a computational system when doing computation about (and by that possibly affecting) its own computation.*
>
> *A reflective system is a computational system which is about itself in a causally connected way.*
>
> *(Maes, 1987)*

Reflective systems are generally structured in two levels: the *base-level* and the *meta-level*. The **base-level** provides the system's functionality. It defines a computational system (i.e. a set of computational elements) that reasons about and acts upon some part of the world, usually called the domain of the system. This level incorporates internal structures representing the domain and a program prescribing how these data may be manipulated. It is often referred as the *base-system* of a reflective system.

On the other hand, the **meta-level**[8] provides the reflective capability. It defines a computational system that reasons about and acts upon another computational system, i.e. the defined in the base-level. The system defined in the meta-level is a system whose domain is another system (i.e. the base-system). Thus, it incorporates structures representing the base-level and a program that manipulates and changes such structures. This is often referred as a *meta-system*.

Both levels, the meta-level and the base-level, are **causally connected**: the structures defined in the meta-level and the domain they represent (i.e. the base-level) are linked in such a way that if one of them changes, this leads to a corresponding effect upon the other (Maes, 1987). In other words, the changes performed by the meta-level on its data structures are reflected somehow in the real system, i.e. the base-level.

In order to observe or change something, this must be represented in a way that a program can manipulate it. This is addressed in the notion of **reification**:

> *The process by which a user program P or any aspect of a programming language L, which were implicit in the translated program and the run-time system, are brought to the fore using a representation (data structures, procedures, etc.) expressed in the language L itself and made available to the program P, which can inspect them as ordinary data. In reflective languages, reification data are causally connected to the related reified information such that a modification to one of them affects the other. Therefore, the reification data is always a faithful representation of the related reified aspect.*
>
> *(Malenfant et al., 1996)*

In other words, **reification** is the action of exposing the internal representation of a system in terms of programming entities that can be

[8] In general, the term *meta-* refers to an artifact that reasons and acts upon another artifact. For instance: a meta-component is a component that acts upon components.

manipulated at runtime. The opposite process is called **reflection**[9], which effects the changes made to the reified entities into the system. However, these definitions may induce to confusion since they do not refer to the base-level and meta-level a reflective system is characterised by. This is taken into account in the definition from Carlos E. Cuesta:

> Reification *is the process that shifts-up an artefact from the base-level to the meta-level, where this artefact will be manipulated.* Reflection *is then the inverse process that shifts-down the artefact from the meta-level to the base-level. Thus, the* reification *and* reflection *processes implement the causal connection among the base-level and the meta-level.*
>
> *(Cuesta, 2002; pp. 88)*

Finally, there are two kinds of operations that can be performed at the meta-level: introspection and intercession. **Introspection** is the ability of a program to observe, and thus reason, about itself. That is, it comprises the operations of a program defined at the meta-level which examine the data structures and program operations of the base-level. **Intercession** is the ability of a program to modify its execution state. That is, it comprises the operations of a meta-level program which change the data structures and program operations of the base-level (see Figure 3.4).

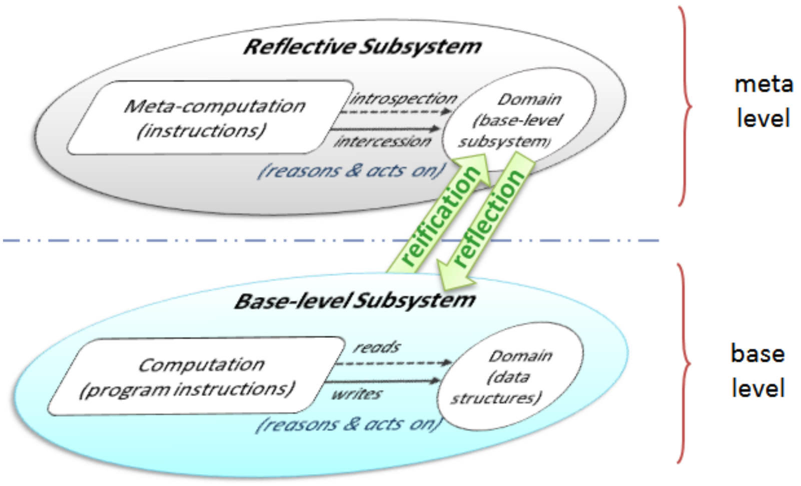


Figure 3.4. Computational reflection: main concepts

[9] Some authors prefer not to use the term *reflection* to define the action of reflecting changes on the base-level, and use the term *absorption* instead. The reason is to avoid confusions with the global term of Reflection. However, we prefer the use of this term to preserve the simmetry of operations, in accordance with (Cuesta, 2002).

The area of computational reflection has quickly spread to different areas: e.g. object-oriented systems (Cazzola, 1998), software architectures (Cazzola et al., 1999a), (Cuesta et al., 2002) or dynamic petri nets (Capra & Cazzola, 2009), among others.

3.6 Dynamic Evolution in Software Architectures

Software architectures (Perry & Wolf, 1992) (Shaw & Garlan, 1996) have become important due to the increasing diversity and complexity of current software systems, which are open, distributed and concurrent. These systems, at runtime, may add new components and interactions as part of their normal behaviour, causing it to diverge from its initial architecture. Because such changes may interact with other parts of the system, it is desirable for the architecture to document and allow reasoning about the changes that can occur during the system execution (Barais et al., 2008).

Software architecture descriptions that include not only the description of fixed (i.e. static) parts, but also the description of changing (i.e. dynamic) parts, are called *Dynamic Software Architectures* (Allen et al., 1998), (Medvidovic & Taylor, 2000), (Cuesta et al., 2001), (Barais et al., 2008). One of the definitions that better captures this nature is the following:

> *Dynamic Software Architectures represent systems that do not simply consist of a fixed, static structure, but can react to certain requirements or events by run-time reconfiguration of its components and connections.*
>
> *(Baresi et al., 2004)*

Several Architecture Description Languages (ADLs) have been proposed for the description and specification of dynamic software architectures (Bradbury et al., 2004). The dynamism provided can be of two kinds: *dynamic reconfiguration*, if the structure of the system is changed, *dynamic evolution of architectural types*, if the types that compose this structure are changed. These kinds of dynamism are described in detail in the following sections.

3.6.1 Dynamic Reconfiguration

The first kind of evolution, called *dynamic reconfiguration*, is a specific kind of evolution that refers to runtime changes that are performed on the structure (i.e. the architecture) of a software system. This kind of dynamism is fundamental for designing systems which often change their structure, such as self-organised systems or mobile systems. It can also be used to implement context-awareness behaviours (e.g. adaptation to different environmental conditions, such as day/night situations in autonomous robots) or to provide fault tolerance support (e.g. replacement of failed components).

The term *Dynamic Reconfiguration* has its origin in the area of distributed systems, as "*Dynamic configuration: the ability to modify and extend a distributed system while it is running, without rebuilding the entire system*" (Kramer & Magge, 1985). Previous approaches only considered *static* configurations of distributed systems: to change some element of a configuration, the entire system had to be rebuilded again. Subsequently, as Software Architecture was consolidating, the concepts of static and dynamic configuration become progressively introduced in ADLs. An example of this is the development of the DARWIN ADL (Magee et al., 1995), which integrated the notions of dynamic configuration presented previously in the area of distributed systems.

Several definitions for dynamic reconfiguration have been proposed, but none of them has been widely accepted yet. Endler and Wei defined dynamic reconfiguration as "*changing part of an application while it operates. [...] Dynamic reconfigurations are expressed only at the configuration level, i.e. as changes to the program's connectivity structure.*" (Endler & Wei, 1992). Jeff Magee and Jeff Kramer also referred to changes on the structure of a system when they used the term dynamic reconfiguration:

> *A feature of an ADL which permits the description of dynamic software architectures in which the organisation of components and connectors may change during system execution [...]. Structural evolution includes changes in both the bindings (connections) between components and the set of component instances.*
>
> *(Magee & Kramer, 1996)*

This definition considers dynamic reconfiguration as a feature of ADLs for describing, or modelling, software systems with a dynamic structure. That is, a feature for supporting *programmed proactive evolution* (see section 3.3.4.2) in the system structure. However, other authors do not constrain dynamic reconfigurations to be programmed, and also consider *reactive* or *ad-hoc reconfigurations* (Endler & Wei, 1992), (Moazami-Goudarzi, 1999), (Wermelinger et al., 2001).

Another definition is the proposed by Carlos E. Cuesta:

> *The term Dynamic Reconfiguration is used to refer, generally, to those changes that are produced [at runtime] in the topology of a composite system (i.e. any alteration of the number and order of nodes and links that define the system).*
>
> *(Cuesta, 2002)*

These definitions have in common that they refer to dynamic changes that are performed in the *organisation/topology* of a system. That is, changes involving the creation and removal of architectural element instances (i.e. components and connectors) and/or the links among them.

Note that these changes affect the set of *architectural instances* of a system, but not the types or specifications that define the internal structure or behaviour of these instances. For this reason, some authors have preferred to use the term *structural dynamism* as a synonym for dynamic reconfiguration, referring to changes at the configuration level, as opposed to *architectural dynamism*, where types can be also evolved (Cuesta et al., 2001). The second level of dynamism is described in the following section.

3.6.2 Dynamic Evolution of Architectural Types

The second kind of dynamism provided in Dynamic Software Architectures, called *dynamic evolution of architectural types*, allows us to change completely the type of an architectural element (i.e. its specification) and its instances at runtime. This kind of dynamism is necessary for building open systems or updating highly available systems. For instance, dynamic evolution can be used to support unforeseen changes, like the addition of new behaviours at runtime (e.g. addition of new tools to an autonomous robot), or the dynamic updating of components to correct malfunctions.

Carlos E. Cuesta et al. proposed the term *architectural dynamism*, referring to the support for the dynamic evolution of architectural types:

> *[Architectural Dynamism] allows the modification of the infrastructure in which structures are defined; that is, the dynamic (re)definition of new component types. Only the latter creates real* new *architectures, so some authors suggest that the term "Dynamic Architecture" should be reserved for this case.*
>
> *(Cuesta et al., 2001)*

That is, the dynamic evolution of architectural types allows the introduction of new, unforeseen architectural types and links, the removal of existing architectural types, or even the modification of such architectural types at runtime. This modification will involve the updating of all the instantiations of an architectural type (i.e. a component type or a connector type) and the migration of their state. In Chapter 7 the concept of *dynamic evolution of architectural types* is defined in more detail (see section 7.2, page 250).

3.6.3 Combining both kinds of dynamism

Both kinds of dynamism are complementary: dynamic reconfiguration acts at the configuration level (i.e. which defines how architectural instances are organised and linked), whereas dynamic evolution of architectural types acts at the type level (i.e. which defines the behaviour and structure –the specification– of architectural types). In certain cases both kinds of dynamism may be needed together: a change in the topology of a software system may require changes in the behaviour of its components, and vice versa.

For instance, a component may be disconnected or removed from the system architecture (i.e. a dynamic reconfiguration operation) in case it is failing. However, since this operation will reduce the available functionalities of the overall system, the current strategies or behaviours must be adapted to deal with the new situation. That is, the emerging system must remove all the services depending on the removed functionality (i.e. a dynamic type evolution operation).

3.7 Conclusions

This chapter has presented the fundamental concepts related to the dynamic evolution of software systems and at the end, it focus on the architectural level. Due to the lack of a widely accepted definition for dynamic evolution, several definitions and alternative terms have been proposed in the literature. This chapter has gathered together the different terms and definitions and has unified them to propose a more general definition of dynamic evolution. In addition, this chapter has also described the state of the art of the main issues that a system that support dynamic evolution should take into account. They are the safe stopping mechanisms (i.e. quiescence and tranquillity), and the updating of stateful artefacts in its different variants.

Other interesting research areas that address some kind of dynamic change management have been introduced: control systems, autonomic computing and computational reflection. Some of the key concepts from these research areas have been used in this thesis to model and support dynamic evolutions: reflection and the autonomic control loop, respectively.

Finally, the different levels of dynamism that can be considered at the architectural level have been presented: the dynamic reconfiguration and the dynamic evolution of architectural types. Both levels of dynamism are interesting because of the degree of flexibility provided for the building of truly open evolvable software systems. So that, these two levels has been addressed in this thesis.

CHAPTER IV

RELATED WORKS

4.1 Introduction

The dynamism in software architectures have been addressed from different perspectives. Some works have focused on the formal specification and description of dynamic reconfigurations (i.e. programmed proactive changes). Others have focused on the low-level mechanisms that provide support for dynamic change (referred also as dynamic updating), and thus providing support to dynamic type evolution. Other works have focused on the design of self-managing systems that are capable of dynamically reorganizing their structure according to some predefined goals. This chapter introduces the most well-known approaches, to give the reader an overview of the different findings and problems that have been dealt with in the design and development of dynamic software architectures.

This chapter is organized as follows. Section 4.2 presents an overview of the different approaches dealing with dynamic evolution. These approaches are categorized in: (i) formal ADLs for reconfiguration, (ii) middlewares for dynamic change support, (iii) self-managed software architectures, and (iv) aspect-oriented management of evolution. Next, section 4.3 presents a comparison among the different approaches, focusing on several attributes such as the level of dynamism, activeness, separation of concerns, or consistency management. Finally, section 4.4 presents the results of the analysis performed over the different approaches.

4.2 Dynamic Evolution Approaches

4.2.1 Formal, Dynamic ADLs for Reconfiguration

In the last decade, several Architecture Description Languages (ADLs) have been proposed for the description and specification of dynamic software architectures (Bradbury et al., 2004), particularly for supporting dynamic reconfiguration.

Generally, these approaches integrate specific reconfiguration primitives in the ADL to describe when and how the system architecture should be reconfigured, i.e. to support programmed proactive reconfigurations. The advantage of using an ADL for describing the programmed dynamism of a system is that an ADL allows rigorously specifying the global architecture of a system, which can be analysed by automated tools. In addition, since several ADLs also provide features that support the automatic generation of parts of a software system, this will facilitate the building of dynamically reconfigurable systems.

4.2.1.1 Process Algebra Formalisms

One of the earliest dynamic ADLs was **Darwin** (Magee et al., 1995). Darwin is a declarative language with an operational semantics based on π-calculus and which allows the hierarchic specification of distributed systems. Although basic components are specified in a programming language, their interface is represented in the Darwin ADL, to allow the specification of the system architecture. Composite components are only specified in the Darwin language, and consist of instantiations of other components and the relationships among them. Dynamic behaviour is defined by means of lazy and direct instantiations. In the former, each component is not instantiated until one of its services is requested. In the latter, components are instantiated directly. However, Darwin only allows component instantiation, but not its removal neither the creation/destruction of links among them.

Leda (Canal et al., 1999) is another ADL based in π-calculus. The language is structured in components, representing system modules, and roles, describing the observable behaviour of components. Attachments define the connections among component instances. Leda also allows the hierarchic specification of systems. Reconfiguration operations are implicitly provided by the use of π-calculus channels: (i) the restriction operator (which is represented by the *new* keyword) allows the instantiation of components (processes), (ii) the transference of channel names among processes allows the dynamic creation of links, and (iii) when a channel name exits from a process then the destruction of instances and links are naturally done. Thus, Leda provides full dynamic reconfiguration support. However it does not provide any tool for supporting reactive/ad-hoc reconfigurations.

Dynamic Wright (Allen et al., 1998) is a language that specifies the behaviour and reconfiguration of a system in a variant of the process algebra CSP. However, since this algebra only describes static configurations, dynamic configurations are not really supported and have to be simulated. All the reconfiguration specifications must be centralized in a global component called *Configuror*, which provides the reconfiguration operations and is the only element that can modify the architecture. The reconfiguration operations that are supported are *new, del, attach* and *detach* for creating instances, deleting instances, linking instances and unlinking. A replacement operation is not provided, so dynamic updating of architectural instances (and state transfer) is not supported.

4.2.1.2 Graph-Based Formalisms

The ADLs presented above are based on process algebras, which are commonly used to study concurrent systems. By contrast, other ADLs have been developed on top of graph-based formalisms: graph grammars are used to specify software architectures and architectural styles, and a graph to represent a specific configuration. Then, dynamism is specified by means of graph rewriting rules to represent the reconfiguration.

The **Hirsh et al.** approach (Hirsch et al., 1998) represents software architectures by means of (hyper)graphs and architectural styles as (hyper)graph grammars. The reason to use hypergraphs and not graphs is to allow the description of multiple interactions. Unlike other graph-based approaches, edges of each graph are components, and nodes represent connectors, ports of communication. Dynamic reconfiguration is modelled by means of graph rewriting combined with constraint solving, which are used to coordinate the dynamic evolution. The language is entirely visual, and does not provide support for programmed reconfigurations: only supports ad-hoc reconfigurations. The approach constrains reconfiguration rules to be context-independent: an edge (i.e. a component) can be only rewritten if its nodes (i.e. its ports) remain connected after the reconfiguration. However, this has the limitation that ports cannot be removed (Cuesta, 2002, pp. 58).

The work of **Wermelinger and Fiadeiro** (Wermelinger et al., 2001), (Wermelinger & Fiadeiro, 2002) combine concepts from category theory, graph grammars and the CommUnity language (Fiadeiro & Maibaum, 1997). A configuration is defined as a graph, which nodes represent CommUnity programs and edges represent instance morphisms, which describe the synchronizations among programs. Reconfigurations are specified through conditional graph rewriting rules that depend on the state of involved components. These rules are defined in a way that is guaranteed the transference of state during replacement (by means of a process superposition morphism) and that components removed are previously in a quiescent state (by following a double-pushout approach, which guarantees that the components are not removed during interactions). Reconfigurations are performed maintaining the architectural style, i.e. conforming to structural constraints. Reconfigurations can be reactive, by means of an external tool provided by the CommUnity language, or programmed. However, although the language is very powerful, its main disadvantage is its complexity.

Baresi et al. (Baresi et al., 2004) describe architectural styles by means of *typed graph grammars:* a type graph defines the elements of a given style and their relationships, constraints define the valid models (by means of cardinalities and OCL constraints), and graph transformation rules represent both communication mechanisms and reconfiguration mechanisms of the considered platform. A configuration that conforms to a given style is represented as an instance graph of the type graph. Reconfiguration mechanisms are modelled using graph rewriting rules, which can be applied to change a graph instance (i.e. a configuration) according to the restrictions defined in its graph type. The authors also used the double-pushout semantics to ensure the consistency between architecture instances. In their example, creation and removal of connections are the only supported reconfiguration operations, so they do not address the problems of quiescence and state transfer. However, the model is so powerful that all kind of reconfiguration operations could be modelled. The characteristic of this approach is that the type graph contains meta-types for both component types and instances (similar to the meta-model of UML). In this way, instance graphs does not only describe the run-time configuration of a concrete architecture, but also its application-specific component types. This allows reconfiguration rules of the style to operate on both the type-level and the instance-level at the same time, necessary to support the dynamic evolution of architectural types.

With respect to algebra-based ADLs, graph-based approaches have the advantage that the consistency of the system architecture is preserved in reconfiguration processes: those reconfiguration operations that violate the architectural style (i.e. their type-graph) are not allowed and cannot be performed. However, graph-based approaches have the disadvantage that their languages are excessively verbose and in several cases only support ad-hoc reconfigurations.

4.2.1.3 Reflection-Based Formalisms

The previous ADLs provide support for dynamically reconfiguring a concrete architecture. However, none of them provide primitives to allow a system to be aware of its actual configuration, and to base its future reconfiguration decisions on this information. This ability is called introspection and was introduced by reflective ADLs.

Of particular interest in this context is **Marmol** (Cuesta et al., 2001), (Cuesta, 2002), a *Meta Architectural MOdeL* for introducing the concepts from Computational Reflection to the field of Software Architecture. It defines the features that an ADL should be provided with to be fully reflective: the distinction among base-level and meta-level, the relationship among these levels (reification and reflection), the notion of meta-component and meta-connector, the notion of meta-space (a set of meta-components that reify a concrete component), and the concept of reflective tower. An example of the application of this model is PiLar.

PiLar (Cuesta et al., 2001), (Cuesta, 2002) is a formal ADL which allows for the representation of hierarchical systems that reconfigure and evolve dynamically. Behaviour is specified by means of constraints, described in a process-algebraic-like syntax inspired in CCS, which is a static algebra. However, the language is provided with reflective capabilities, making the ADL fully dynamic. This is demonstrated by the formalisation of its semantics in π-calculus. Thus, Carlos E. Cuesta demonstrates his hypothesis that the functionality obtained with any static language or algebra (such as CCS) combined with reflection is as powerful as other native dynamic language or algebra (such as π-calculus). Every component instance can be *reified*, and be manipulated by meta-components, which are capable of doing full introspection and intercession on the component instances. In this way, any reconfiguration operation can be performed: the explicit creation/removal of architectural instances and/or links. This ADL supports implicitly the stratification of an architecture in meta-layers: meta-components can be also reified and be modified by meta-meta-components. The reflective nature of PiLar allows both describe dynamic reconfigurations and dynamic evolution of architectural types. However, one of its disadvantages is that it does not isolates syntactically base-level from meta-level descriptions, which difficults the separation of the reconfiguration concerns from the base functionality.

Archware ADL (Oquendo et al., 2004) is a formal language based on π-calculus which provides reflective capabilities by means of hyper-code (Kirby et al., 1992) and the concepts of dynamic system composition and decomposition. π-calculus is used for specifying hierarchical executable architectures: components are represented by behaviors and the interaction between components via channels is represented by communication on connections. The language provides specific operations for reflect and reify. Evolution is effected by decomposing the selected part of the system, reifying the components, applying a transformation, and finally reflecting the code. Decomposition is provided by means of a *decompose* operator, which breaks up a composite behaviour (e.g. a composite component or the system architecture) into its constituent behaviours (Balasubramaniam et al., 2004). These constituent behaviours, on decomposition, are not suspended: each continues to execute until it reaches its reduction limit, which is typically when it would have interacted with another behavior. Then, these constituent behaviours are reified through *hyper-code* (Mickan et al., 2004), which allows the interactive introspection and modification of code at run-time. Finally, changes are reflected by means of the *compose* operator, which defines a composite behaviour as as the composition of the behaviors of its subunits. Thus, not only the dynamic reconfiguration of architectures is supported, but also the dynamic evolution of types that are defined. However, only ad-hoc reconfigurations and changes were supported in the initial version of Archware ADL. Further extensions have addressed the definition of proactive changes (Balasubramaniam et al., 2005).

4.2.2 Systems for Dynamic Change Support

The approaches presented in the previous section are focused in the specification of dynamism in software architectures for its analysis or simulation (e.g. Darwin, Leda, Wermelinger *et al.* or Baresi), or the description of proactive dynamism (e.g. Darwin, Leda, PiLar or Archware). However, these approaches only cover the specification of dynamism, but not how this dynamism is supported: i.e. how the state is transferred on dynamic updatings, how the elements are stopped for being evolved, or how the structures are changed dynamically.

This section introduces the works that have dealt with such low-level details, which are very important to completely support the execution of truly dynamic evolvable systems. These works, generally called systems for dynamic updating or dynamic change support, focus on the modification of runtime structures. A more extensive analysis of the state of the art can be found in (Vandewoude, 2007) and (Hammer, 2009).

4.2.2.1 Procedural and Object-Based Techniques

The earliest works that addressed dynamic change support were characterized by a fine level of granularity: the elements subject to changes were statements, procedures or modules. Segal and Frieder (Segal & Frieder, 1993) reviewed the first approaches (around the late 1970s and earlier 1980s) to support dynamic program updating. Such earlier works were mainly based on supporting the evolution of procedure-oriented programs, with little emphasis on the concepts of types and instances. Generally, most of the techniques proposed only differed on how the indirection was managed. An interesting review of the wide range of techniques used to recompose sotware systems at runtime can be found at (McKinley et al., 2004), or its extended version (McKinley et al., 2004a).

Fabry's work (Fabry, 1976) was one of the first to investigate the updating of (abstract) data types and the migration of their instances. His work provided deferred updates, using version attributes to distinguish the out-of-date instances. The main mechanism used was the use of indirections at code segments: all the calls to the old code were updated with the address of the new code segment. Thus, running processes could continue executing the old code, whereas new processes would start executing the new code. However, this means that not only the old code segment is modified, but also the calling programs, which results in invasive evolutions. The benefit of current approaches is that interactions among processes are made explicit and separated from the functional behaviour, thus facilitating their evolution. Another limitation of Fabry's work is that external interfaces could not be modified.

With the expansion of object-oriented languages and frameworks, the distinction among types and instances (e.g. classes and objects) become evident, and also the need for dynamically updating both. **JDRUMS** (Andersson et al., 1998), (Ritzau & Andersson, 2000) provides transparent dynamic updating features to Java programs, by means of a modified Java Virtual Machine. It extends the Java class loader (i.e. the Class and Object Java meta-types) to include a link to the class (or object, respectively) that replaces it. When a class is dereferenced (i.e. its reference is popped out from the running stack), JDRUMS checks if it has been updated and then returns the last version. Object updates are performed when they are dereferenced; their old, internal state is converted to the structure of the new class. Thus, different versions of the same class, as well as their objects, can coexist simultaneously: asynchronous evolution is supported. A set of tools are provided to drive the updating process (i.e. support for reactive changes), to help in the definition of state transfer methods, and to help in the propagation of changes to multiple Java Virtual Machines running in a distributed environment. JDrums also provides an interface to allow the running application to drive the dynamic evolution process by itself (i.e. support for proactive changes). The consistency of updates is preserved by delaying the updating of objects until they are not active: that is, any method is being executed.

Other works have also addressed the dynamic evolution of Java programs by extending the default class loader, such as **Malabarba et al.** (Malabarba et al., 2000) and **Wang et al.** (Wang et al., 2006). In these works, the updating of a class is delayed until no methods of this class are active. Then, the old state is transferred to the new component by using the reflection mechanisms provided by Java (if no new data structures have been introduced). Thus, this guarantees the consistency of the system before and after the updating process. However, the evolution has important constraints: a new class version must preserve both the attributes (i.e. the state structure) and methods (i.e. the interface) of the version it is going to replace. This is not always possible when substantial changes need to be introduced. In addition, both approaches perform a synchronised dynamic evolution, not suitable for distributed systems.

4.2.2.2 Dynamic Weaving Techniques in AOP

During the last years, there has been a growing interest in using Aspect-Oriented Programming (AOP) techniques to address the adaptation of applications. In the context of dynamic evolution, the most interesting techniques are those that support *dynamic weaving models*, i.e. the support for runtime weaving and unweaving of aspects to an application. This allows the evolution of existing code by adding or removing aspects. A comparative analysis among four Java-based dynamic weaving systems (i.e. AspectWerkz, JBoss, PROSE, and Nanning) is presented in (Chitchyan & Sommerville, 2004). Next, other relevant works on the topic are described.

Rapier-Loom.NET (Schult & Polze, 2003) supports the dynamic addition and removal of aspects, but weaving definitions are defined inside aspects, thereby losing their reusability. Rapier-Loom.NET acts at the type-level: aspects are added to the base code and then all the instances benefit from the new functionality. The weaving is achieved by dynamically creating a subclass and overriding the crosscutted methods with their woven counterparts. That is, the weaving of aspects is performed only on object creation. The consequence is that the behaviour provided by a new aspect would not be provided to a running instance unless this instance is destroyed and recreated again.

EOS (Rajan & Sullivan, 2003) is another dynamic aspect-oriented approach which also defines weavings inside aspects. EOS may act at the type-level (i.e. adding aspects to a class that will impact all their instances) or at the instance-level. In this case, an aspect is only weaved to instances that satisfy a certain criteria or trigger an event. This is implemented as a mediator pattern. In this sense, EOS supports the addition or removal of aspects at any time. EOS provides reflective information at *join points* to evaluate at runtime where an aspect should be weaved or not.

None of the approaches mentioned above takes into account the emerging relations that result from the aggregation of various aspects at the same point of the base code. However, **JAsCo** (Suvée et al., 2003), (Vanderperren et al., 2005) provides an expressive language that permits the definition of relationships among aspects. JAsCo integrates AOP and CBSD. It introduces the concept of connectors for the weaving between the aspects and the base code, which allows for a high level of aspect reusability. The weaving of aspects is based on a preliminary insertion of a stub or a trap on the base code that will then make appropriate calls to the JAsCo runtime infrastructure. The insertion of traps is done through the debugger interface of the Java Virtual Machine. When a trap on the base code is executed, the JAsCo runtime infrastructure looks for registered connectors (that implement aspect join points) and redirects the execution flow to aspects. This is the main disadvantage of the approach: the dynamic weaving is referential but not inclusive, and requires an infrastructure continuously supervising the execution of the entire system.

PROSE was one of the first platforms that tackled the problem of dynamic AOP (Popovici et al., 2002). It has evolved through three different versions based on different forms of interception and weaving. The first version (Popovici et al., 2002) was intended to demonstrate the potential of dynamic AOP. It used the Java Virtual Machine Debugger Interface event notification mechanism to convert join-points into stop points. Once the application had been stopped, aspect code was executed externally, but with access to the context where it was being executed (e.g. calling parameters for methods). The second version of PROSE (Popovici et al., 2003) extended this model by giving the option of, instead of using the debugger, using the baseline Just-In-Time compiler. The idea was to weave hooks (i.e. traps) into the application at native code locations that correspond to all potential join-points (e.g. method definitions). When executed, the hooks determine whether an aspect needed to be invoked for that particular join-point. The last version of PROSE (Nicoara et al., 2008) further explores the way aspects can be weaved to code and provides additional alternative weaving mechanisms to increase the flexibility of the system. In essence, PROSE uses code interception and redirection to introduce new code (i.e. aspects) at the method level. All the changes are reversible: aspects can be inserted and withdrawn at any time.

In general, all the dynamic AOP approaches support the dynamic introduction and weaving of aspects through an interactive tool. Therefore, reactive changes are supported. Proactive changes are also supported, although they are defined in a different way with respect to other, non aspect-oriented approaches. Proactive changes in AOP approaches are defined by means of weavings and pointcuts: they define *when* and *where* the aspect code (i.e. the code to be executed instead of the base code) must be introduced at runtime. For instance, a pointcut usually executes aspectual code when a certain method is executed and a condition applies. However, this proactive support is limited: complex ECA conditions cannot generally be defined.

Another advantage of dynamic AOP approaches is that their evolution mechanisms (i.e. the dynamic weaving mechanisms) are also separated from system functionality. These mechanisms are added in a second stage, either at the source code level (e.g. by means of libraries), or at the binary code level (e.g. by means of bytecode traps). If the code of the original system is changed, the AOP parsers can introduce again the traps needed to support the dynamic weaving of aspects.

However, the type evolution support provided by dynamic AOP approaches is limited. The nature of AOP approaches is to *add* or *interleave* crosscutting concerns into the code of an application (i.e. the base code). That is, the base code is only *extended* or *replaced* with aspect code, but it cannot be removed (only deactivated). AOP approaches can be only used to replace the body of methods, but the other elements of a class (i.e. public interface, attributes, available methods) must be preserved. For this reason, current dynamic AOP approaches should not be considered to perform exhaustive refactorings of systems. In addition, since existing data structures cannot be changed, issues such as state transfer or the safe stopping of instances are not considered.

4.2.2.3 Component-Based Dynamic Frameworks

Dynamic evolution features have also been considered in the development of several component-based models and frameworks. This section introduces the most extended component-based frameworks, focusing on the features they provide for supporting dynamic evolution. Most of the component-based frameworks described here are publically available, although some of them only are research prototypes.

The **Simplex** architecture (German-Rivera et al., 1996), (Sha et al., 1996) supports dynamic updating of commercial-of-the-shelf (COTS) components, with special emphasis on fault-tolerance. Each component internally is composed by four units: (i) the *baseline unit*, which provides the functional behaviour of the component; (ii) the *new unit*, which acts as a placeholder for updates of baseline units; (iii) the *safety unit*, which implements a safety controller, system performance and monitoring functions; and (iv) the *module management unit*, which performs process management and drives upgrade operations. The approach mainly employs a *n-version scheme*: instead of replacing a component by its update, both the old version (i.e. the baseline unit) and the new version (i.e. the new unit) are run in parallel, until the safe operation of the update is confirmed. Input is directed to both the old and the new version, which allows the new version to absorb information from its environment. However, the output of the new version is only used for monitoring purposes. The system compares the output of both the old and the new version: if the new version behaves correctly according to a predetermined user-specified metric, the system then confirms the update and removes the old version. If the update is unsuccessful, rollback is used to return to the original state. Dynamic reconfiguration of the system (i.e. changing connections among modules) is addressed by means of a publish-subscribe approach, carried out by the module management unit when updates are confirmed. This work is fairly abstract and only reports on the concepts used (such as a brief description of state transferal). Changes are supported by a set of external tools, which are not detailed (similarly to other approaches). Dynamic type evolution is partially supported, since the approach does not consider the propagation of updates to other copies of the updated module.

ArchJava (Aldrich et al., 2002) is an extension of the Java programming language that introduces components directly into the language, by means of its own compiler. The authors claim that this allows tight coupling of an architecture and implementation of component-based applications and prevents inappropriate modification of the architecture at runtime. Only basic reconfigurability is provided, component creation and connection, but does not allow explicit component removal. Removals are implicitly done when all the connections to a component instance have been deleted. The framework does not provide mechanisms for state transfer, quiescence or versioning.

OpenCOM v2 (Coulson et al., 2004), (Coulson et al., 2008) is a reflective component model that provides provisions for reconfiguration. It is based on Microsoft's COM component model, and the main programming language is C++. Components are deployed at runtime into environments called capsules (it could be seen as a composite component definition) which provide operations for dynamically loading/unloading of components and also binding/unbinding of interfaces and components. Dynamism is provided by a set of *reflective meta-models*, which permit different system aspects to be programmatically inspected, adapted and extended at runtime. These models are: the *architecture meta-model*, which exposes the compositional topology of the components in terms of a causally-connected graph structure; the *interface meta-model*, which allows the exploration of interface types at runtime and the dynamic invocation of (dynamically discovered) interface instances at runtime;

and the *interception meta-model*, which allows the dynamic adaptation of components, by means of interceptors at bindings between interfaces. By requiring components to implement special interfaces, the provision of reconfiguration-relevant code in the components is ensured. For ensuring safe reconfigurations, quiescence has been recently introduced (Pissias & Coulson, 2008). The component framework is very technical, and most of the problems (e.g., how a connection can be reassigned) arise from details of the actual development. The advantage is that this framework is publically available for research purposes.

Draco (Vandewoude et al., 2003), (Vandewoude, 2007) is an extensible and modular component-based framework which supports the dynamic reconfiguration of systems, as well as the runtime replacement of components. Draco is a middleware platform which is part of a set of tools: a custom component language, a pre-processor which translates this language into standard Java, an Eclipse-based environment that assists the programmer with component development and the component framework, Draco. It is implemented in Java and supports four primitives for changing the composition of systems at runtime: loading, unloading, connection and disconnection of components. Among the different features of the component model, one of the most interesting is the transactional execution of processes and the asynchronous execution support. The framework was designed to be easily extensible through extension modules. One of the most interesting module is the *Live Update Extension* module: it provides support for safe stopping of components (tranquillity *and* quiescence, see section 3.4.1.2), and for the transference of state when updates are performed (see section 3.4.2.3). Among the other frameworks, Draco is the only one that implements advanced algorithms for the safe stopping of running systems and the updating of their components.

Fractal (Bruneton et al., 2004) is a hierarchical component model which provides sharing capabilities (to model resources and resource sharing while maintaining component encapsulation), introspection capabilities and reconfiguration capabilities. The Fractal component model distinguishes two kinds of components: primitives (i.e. simple components, which contain actual code), and composites, which are only used as a mechanism to group components into a whole. Fractal provides an XML-based ADL that provides constructs to specify component types, primitive templates and composite templates. Fractal ADL specifications are parsed and instantiated to a particular executing platform. One of the most widely used implementations of Fractal is *Julia* (Bruneton et al., 2004), a publically available Java library that enables the specification and manipulation of components and architectures at runtime. Each component consists of a *controller*, which manages all the interactions of the component with the outside. This controller provides a set of *control* interfaces that provide reflective capabilities: service discovery and lookup (i.e. introspection), creation and destruction of bindings, addition and removal of sub-components, etc. These interfaces allow both ad-hoc and limited programmed reconfiguration. Programmed reconfiguration is limited because the ADL only allows expressing a single instantiation of a system, indicating how its components are instantiated and interconnected, but not different configurations. Another drawback is that safe stopping and state transfer is weakly supported: they are mentioned, but barely discussed. However, since Fractal is open and extensible, other approaches have extended the component model to increase the level of dynamism provided. For instance, **WildCAT** (David & Ledoux, 2005) allows the exploration of the application's execution context at runtime. WildCAT provides a context model that changes dynamically to reflect changes in the actual execution environment: attribute values or resources can change, appear or disappear at any moment. These modifications generate external events which can be captured by a reconfiguration policy. This is very useful to implement context-aware systems.

SOFA 2.0 (Bures et al., 2006), (Bures et al., 2007) is a hierarchical component model whose design has been influenced by the experience obtained with an earlier system for dynamic updating, called SOFA/DCUP (Plasil et al., 1998). The model provides a text-based ADL called Component Definition Language (CDL), which allows specifying the communication among SOFA components and embeds a process algebra called behavior protocols to express the behavior of each component. A component is represented by a set of interfaces, both provided and required, and which determine the component's type. Components can be primitive (it is defined by binary code), or composite (it is defined by a set of subcomponents and interconnections among them). An interesting characteristic of the model is that components include a specification of behaviour in terms of the event traces determined by the secuencing/parallel method call acceptance on the provided interfaces and their reactions on the required interfaces. This constrains the set of admissible traces of a component and allows the analysis of behaviour. The other component models, such as ArchJava and Fractal, only check that connected ports provide and require services with compatible signatures. In SOFA 2.0 reconfiguration is inherently supported: component instances can be dynamically created and removed, by means of the dynamic detaching and attaching to connectors. These functionalities are encapsulated in controllers, which are the control part of a component (like in Fractal). SOFA 2.0 is one of the few frameworks where components can request reconfigurations, focusing on the hierarchical structure. State transfer for dynamic updates is not discussed but, since version management is supported, passive partitioning may be used (i.e. coexistence of old versions with newer versions). The framework also provides a tool to interactively manage a running SOFA system.

4.2.3 Self-Managed Software Architectures

Self-management is presented as a way by which systems could be scalable, support dynamic composition and rigorous analysis, and be flexible and robust in the presence of change (Kramer & Magee, 2007). Dynamic evolution in this context is essential to support this kind of flexibility.

There are two main engineering approaches realizing self-managing systems, which differ on how the interaction patterns of the system are managed. On the one hand, top-down approaches rely on a centralized representation/model, which describes the relations among the different elements of the system. Self-management is guided by this model: (global) decisions are taken upon this model and changes are applied on the system globally. An example of top-down approaches are self-adaptive systems (Oreizy et al., 1999), which are based on an architecture model and a set of (high-level) goals to guide the adaptation process.

On the other hand, bottom-up approaches are fully decentralized: interactions are managed locally by the elements of the system. Thus, self-adaptability emerges from the local adaptation decisions taken by each component. An example of bottom-up approaches are self-organising systems (Serugendo et al., 2006), which are based on algorithmic functions to guide the (local) adaptation process. These systems, usually of a distributed nature, are composed of several autonomous instances, which run concurrently and organize themselves according to different criteria.

This section describes some of the representative works that realise each engineering approach: top-down or self-adaptive approaches, and bottom-up or decentralized approaches.

4.2.3.1 Top-down approaches: Self-Adaptive Systems

In parallel with the development of techniques for specifying and supporting the dynamic reconfiguration of software systems, several works have addressed the design and development of *self-adaptive systems*:

> *A* **self-adaptive system** *is one that: (i) is capable of adapting its structure at runtime in response to changing operating conditions or user requirements, and (ii) relies on a centralized model of the system, which defines the high-level goals that guide the changes.*

Self-adaptive systems perform *dynamic adaptations* on the system, i.e. dynamic reconfigurations involving *non-intrusive* changes or replacements, but not dynamic evolutions on their types. These changes are applied at the architectural level, following a top-down perspective, and are guided by a set of (high-level) goals and a centralized model of the system (generally an architectural model).

The first comprehensive self-adaptive conceptual framework was presented in 1999 by **Oreizy et al.** (Oreizy et al., 1999). This framework defined an adaptation management lifecycle consisting of three tasks: (i) monitoring and evaluation of observations from the system execution (e.g. performance monitoring, safety inspections, constraint verifications, ...); (ii) planning of changes, that accepts evaluations, defines appropriate adaptations, and constructs a plan for executing that adaptations; and (iii) deploying of change descriptions, that propagates the sequence of changes, components implementing such changes, and possibly new observers or evaluators to the implementation platform. This adaptation lifecycle could be fully autonomous or have humans in the loop. The framework is characterized for being architecture-based: all the changes are formulated, and reasoned, in terms of the architecture. For this reason, an architecture model of the system is maintained explicitly on the implementation platform; changes on this model are reflected in modifications to the application's implementation, while ensuring that the model and the implementation are consistent with one another. Monitoring and evaluation services observe the application and the operating environment and feed this information to the adaptation management cycle.

The architecture-based self-adaptive framework proposed by Oreizy et al. has several similarities with architecture-based reflective frameworks. **Cazzola** *et al.* (Cazzola et al., 1999), (Cazzola et al., 1999a) were one of the first authors to propose a reflective framework to address the reconfiguration of software architectures. They introduced the term *Architectural Reflection* as "*the computation performed by a system about its own software architecture*". Architectural reflection is provided by reflective mechanisms that exploit the architecture of a system as the application domain. The software architecture is *reified* by these reflective mechanisms (i.e. the architecture is made explicit, observable, and the system controllable through its architecture), decomposed into topology and strategy and manipulated, respectively, by two meta-level components, called *topologist* and *strategist*. These meta-level components plan and force the application's reconfiguration through the manipulation of the reified architecture.

These earlier works on self-adaptive architectures were merely conceptual and did not address implementation details: e.g. the specification of reconfiguration goals or policies, the coordination and propagation of changes, the management of the application consistency, etc. However, these works were very influential to the advance in the development of self-adaptive systems, and several works followed (Kramer & Magee, 2007), (Cheng et al., 2009), (Cazzola, 2009). Next other interesting self-adaptive frameworks are introduced.

The **K-Component model** (Dowling & Cahill, 2001) is a reflective framework for building self-adaptive systems. Self-adaptive tasks (i.e. observation, reasoning and act) are performed on a reification of the architecture being managed. This reification is modelled as a typed, directed configuration graph, where interfaces are the vertices, components are the type labels, and connectors are directed edges. This configuration graph is automatically built as a dependency graph from component definitions and the connections among them. Reconfiguration planning, validation, and execution of changes are performed over this configuration graph by means of *adaptation contracts*. An adaptation contract is a reflective program that defines: (i) architectural constraints, (ii) adaptation events, and (iii) conditional statements that associate the occurrence of adaptation events with a set of reconfiguration operations. If adaptation is required, a component can be removed from the configuration graph and another component, exposing the same interface, can be swapped in. A single meta-entity, called the *configuration manager*, provides the execution environment for adaptation contracts, supports the dynamic loading and unloading of adaptation contracts, and encapsulates all the actuators and reflectors that act on the running system. This model uses the reconfiguration protocol proposed by Wermelinger (Wermelinger & Fiadeiro, 2002) to reach a safe state for reconfigurations. The K-Component model also considers the transfer of state among component when performing updates, by means of a user-defined constructor. The main disadvantage is the inability of the framework to accept new types in the configuration graph, since the graph is built statically at compile-time.

Dashofy, van der Hoek and Taylor (Dashofy et al., 2002) describe an infrastructure to build self-healing, architecture-based software systems. Their approach consists of dynamically generating a repair plan based on the differences among the actual state of the system and the configuration that the system should have. Repairs are done at the architectural level. The repair plan is generated by a fault-detection/planning agent that monitorizes the running system, detects possible faults, and proposes possible repairs. The architecture of the system and the set of proposed changes are described in xADL 2.0, an extensible ADL (Dashofy et al., 2001). When a fault is detected, this agent generates a new architectural model that includes a set of proposed repairs and which is analysed to detect whether the new model is valid or not. If the new model is valid, the repair plan is executed by the *architecture evolution manager*, a meta-entity that invokes the needed low-level evolution services provided by the runtime infrastructure. The runtime infrastructure performs the required changes in the whole system, but the details about this infrastructure are not provided.

The **Chisel** framework (Keeney & Cahill, 2003) is a reflective system for dynamic evolution in a policy-driven, context-aware manner. It is based on the same principles of aspect-oriented development: an object only keeps its core functionality, and the non-functional behaviour is separated into multiple possible behaviours, called *meta-objects*, that can be reused by other objects. As the environment, user context or application context change, objects will be adapted to use different behaviours, driven by a human-readable declarative adaptation policy specification. This is performed by a meta-level adaptation manager that coordinates the whole adaptation process (monitoring, planning and reflecting changes). Although the framework is not architecture-based and focuses on changes at a small granularity level (i.e. objects), the contribution of the work is its support to unanticipated changes and the definition of context-aware policies.

RAMSES (Cazzola et al., 2004) is another framework that exploits reflection to provide applications with self-adaptative capabilities. It has two logic levels: the base-level, where the system to be adapted runs, and the meta-level, where two meta-objects, called *evolutionary* and *consistency checker*, take care of planning and validating the evolution of the system. Evolutions and validations are driven by a set of ECA rules, and are executed over a reification of the base-level. The reification of the base-level is modelled as UML diagrams. These diagrams are encoded as XMI models, so they can be processed at runtime. The two meta-objects perform changes on these models, which are reflected back through code instrumentation techniques. Although this approach is not architecture-based, the techniques that are proposed to manipulate models at runtime are also interesting, and could be easily applied to architecture-based self-adaptive systems.

The **Rainbow** framework (Garlan et al., 2004), (Cheng et al., 2005) provides a reusable infrastructure for self-adaptive systems together with mechanisms for the specialization of that infrastructure to specific systems. Rainbow integrates an architectural model of the system in its runtime system, which is used by a control loop for self-reconfiguration purposes. The control loop is composed of four modules: i) a model manager, that handles and provides access to the application's architectural model; ii) a constraint evaluator, that supervises the model periodically and triggers adaptations if a constraint violation occurs; iii) an adaptation engine, that determines the set of reconfiguration actions to be performed; and iv) an adaptation executor, that reflects the changes on the application. The monitoring of the system is performed from outside, with probes measuring data and gauges aggregating these data to provide decision criteria for reconfiguration. Reconfiguration is triggered by rules that are based on the output of gauges. Dynamic changes are carried out by external effectors, but their behaviour is not detailed. The granularity of changes are very coarse (e.g. web clusters), and a great emphasis is placed on maintaining the architectural style.

Plastik (Batista et al., 2005) is a meta-framework for reconfiguration that bridges the gap among a high-abstraction level language, Armani ADL (Monroe, 98), and a component-based middleware, OpenCOM (Coulson et al., 2004). Armani is an extension of the ACME ADL (Garlan et al., 1997) that allows the description of architectural constraints over ACME architectures, but that does not support the specification of dynamic reconfigurations. Plastik extends the ACME/Armani ADL with statements for describing dynamic reconfigurations: reconfiguration triggers, the removal of components and links (creation of components and links is already covered by standard ACME), and the existence of dependencies among components. Dependencies are introduced to support the fact that, in some cases, the dynamic instantiation/destruction of components is dependent on the creation/destruction of other components. The execution support for these reconfiguration statements is provided by OpenCOM, a component-based middleware that provides reflective capabilities, thus enabling the dynamic reconfiguration of its elements. Thus, software systems can be described and analysed at a high abstraction level, but also executed directly on a runtime platform. Dynamic reconfigurations can be described and executed both reactively and proactively, by means of a centralized *system configurator*. This configurator is divided in two levels: *an architectural configurator*, responsible for accepting and validating reconfiguration requests at the ADL level, and *a runtime configurator*, responsible for managing the OpenCOM runtime level. There is one instance of the architectural configurator in the whole Plastik system, and one instance of the runtime configurator for each deployed OpenCOM component frameworks. In this way, high-level reconfiguration specifications are centralized in the architectural configurator, and their management is delegated to each OpenCOM runtime configurator.

Ayed and Berbers (Ayed & Berbers, 2007) describe a policy-driven framework to dynamically change CORBA component-based applications. This framework extends both the execution and deployment models of CORBA by introducing new entities and adaptation interfaces in the containers of components. The architecture of the system is represented as a graph, which define components and connections that can be either obligatory or optional. Obligatory components and connections are deployed (or created, respectively) for all the possible application contexts. However, the existence of optional components depends on the application context and requires the specification of an existence condition. In the same way, optional connections can only be materialised if the two components that they link also exist. The deployment plan is extended with the description of variability (i.e. obligatory and optional components and connectors), and is context-aware. The prototype proposed supports the safe stopping of component instances and the state transfer (only if data structures are compatible). This functionality is separated in three modules: (i) *context manager*, which is responsible for the collection, storage and reasoning of context information; (ii) *adaptation manager*, which decides the reconfiguration activities to be performed according to the context information and which analyses the dependencies among other instances; and (iii) *consistency manager*, which stops, blocks and transfers the state of component instances for reconfiguration. These modules are provided in a centralized way.

As a summary, self-adaptive approaches are generally characterized by: (i) supporting dynamic reconfiguration; (ii) the separation of dynamic change policies or goals from system functionality; (iii) supporting proactive changes (generally defined by ECA policies inside the self-control loop); and (iv) providing some degree of self-awareness, required to trigger and drive changes. The main advantage of self-adaptive approaches is that all the code related to the management of change (i.e. the reconfiguration policies) is centralized in a single place (i.e. a *configuration manager*) that eases its maintenance. Another advantage is that it makes easy the coordination among different reconfiguration policies. Since the *configuror* has a complete or global view of the system, it can analyse the conflicts that may emerge on other subsystems when applying a change, to prioritize one instead of others (Raheja et al., 2010).

However, the use of a centralized configuration manager also introduces important issues for the development of medium-sized distributed systems or large-sized centralized ones:

- **Poor scalability.** The use of a centralized entity for controlling all the changes of a system is not scalable. The larger the system, the more complex and less maintainable the configuration/adaptation manager would be, since the scope that the manager must supervise will increase proportionally. This will also impact on the time required to analyze the system and take reconfiguration decisions.

- **Single point of failure**. If the configuration manager fails, the overall system would also lose its ability to reconfigure and repair itself. This poses also a security risk: in case the security of the configuration manager is violated, the architecture of the whole system (i.e. all the subsystems) would be available to the attacker, who could change the entire system.

4.2.3.2 Bottom-up approaches: Decentralized Architecture-Based Systems

In the case of large systems or distributed systems, decentralized models are preferrable. Several decentralized models have been proposed in the area of multi-agent systems and self-organised systems (Serugendo et al., 2006), (Santiago-Perez et al., 2009), which could be considered for addressing the building of self-reconfigurable architecture-based systems. Generally, in *self-organised systems* interactions are managed locally by the elements of the system:

> *A* **self-organized system** *is one that: (i) is capable of reorganizing its structure at runtime in response to changing operating conditions; (ii) relies on a fully distributed, decentralized model: it is composed of autonomous nodes which run concurrently and organize themselves according to local decisions.*

Thus, self-adaptability emerges from the local adaptation decisions taken by each component. An example is the work of (Rogers et al., 2009), which is based on algorithmic functions to guide the (local) adaptation process.

In the area of software architecture, few works have addressed the management of change from a decentralized perspective. Some formal ADLs provide implicit support for the description of decentralized reconfigurations: these categorized as *smart components* in (Cuesta et al., 2001). ADLs allowing the definition of smart components are those that provide components with primitives to reconfigure the system architecture (or at least part of it): e.g. Darwin (Kramer & Magge, 1985), LEDA (Canal et al., 1999) or the categorical model of Wermelinger (Wermelinger et al., 2001). In this way, components are free to reconfigure the architecture to which they belong. However, the disadvantage is that reconfiguration specifications are spread among the different components, thus decreasing maintenance of such specifications and making difficult to see the conflicts among the different reconfiguration specifications.

The work from **Georgiadis, Magee and Kramer** (Georgiadis et al., 2002) was one of the first in describing a system with a completely decentralized change execution infrastructure. Architecture specifications are modelled in Darwin

(Magee et al., 1995), but described in Alloy (Jackson, 1999) to express structural constraints and benefit from existing Alloy analysis tools. Each component is packaged with a *configuration view* and a *component manager*. The former is a directed graph of the overall system architecture, and the latter is the maintainer of the configuration view and the manager of the component implementation. Events are used to communicate component managers together with: the binding/unbinding of ports, addition/removal of components, the failure of components, or the modification of component attributes. Architectural constraints, which are deployed in each component manager, guide how components should be bound to others. Reconfiguration is passively driven by constraint satisfaction: every time the configuration view changes, architectural constraints are re-evaluated to detect if a reconfiguration is needed. The disadvantage is that reconfiguration rules are deployed together the management mechanisms, in the component manager. The major issue of the approach is its low scalability: it requires a total order broadcast to maintain the consistency of the distributed configuration views.

The **K-Component model**, in its second version (Dowling & Cahill, 2004) proposes a decentralized architecture-based model for building self-adaptive systems. A K-Component is defined as a runtime framework where components and connectors can be deployed. Each K-Component contains a configuration manager that reifies the architecture defined in the K-Component and provides adaptation contracts to manage this (local) architecture. There is no explicit representation of the system-wide architecture: it is partitioned amongst the K-Components of the system. Each configuration manager exchanges with its adjacent K-Components the information related to the remote components that are connected to its connectors: the reification of remote components, feedback events, and their states. Adaptation contracts thus operate on a K-Component by reasoning about adaptation conditions using either the local information or the shared remote information. Adaptation contracts can be defined using ECA rules (which are programmed at design-time), or using Collaborative Reinforcement Learning techniques. The latter is used to establishing and maintaining system-wide properties in a decentralized system.

The **Evolver-Producer model** (Morrison et al., 2007) is a conceptual framework for the description of evolving systems with a decentralized nature. The framework is based on the concept of locus, producer and evolver. *Locus* is the context where a system, subsystem, or element, may change or evolve. A locus is structured in two functional units: a *Producer*, which carries out productive functionality (i.e. taking input and producing output), and an *Evolver*, which manages the evolution of the locus. The Evolver monitors the Producer, the environmental stimulus, or its own feedback, and uses this information to drive the evolution of the locus to obtain a new version of the Producer or even a new version of itself. These concepts can be recursively applied: both the Evolver and the Producer may be internally composed of an evolver-producer pair, and so on. Thus, each locus is constructed in a system as a mini-control system with the producer as the process and the evolver as the controller (see section 3.5.1). Moreover, the evolver-producer pair model allows loci to be aggregated and structured hierarchically, that is, this does not necessarily imply encapsulation but rather the nesting and aggregation (statically or dynamically) of change contexts. An example of how the Evolver-Producer model can be applied for the building of architecture-based autonomic systems is described in (Balasubramaniam et al., 2005), using the Archware ADL (Oquendo et al., 2004). However, this model does not address the problems that arise on the coordination of different change contexts. Both proactive and reactive changes are supported: the Evolver defines proactive changes, and hyper-code technology (Mickan et al., 2004) supports reactive changes. Regarding the implementation details of change mechanisms, the authors describe some of the techniques that can be used to implement their approach in Java. For instance, state transfer is delegated to the programmer, who must provide a constructor to import the previous state. Although their work describe how new types can be generated at runtime (by using dynamic compiling techniques), their work do not detail how the updated types are bounded to the existing system, replacing the old types. For instance, running instances are not safely stopped: they are only *blocked* when they try to interact with a stopped connector. This may leave the system in an inconsistent state (see section 3.4).

The **Adapt-Medium** approach (Phung-Khac et al., 2008), (Phung-Khac et al., 2010) is an architecture-based approach that deals with the runtime adaptation of distributed applications. The approach relies on the concept of *adapt-medium*, a logical aggregation of local adaptation managers that collaborate together to perform a runtime adaptation. The set of adaptations that can be performed at runtime are defined during a model-based development process as a set of design alternatives, or architectural variants (similarly as performed in product-line approaches, e.g. (Gomez & Ramos, 2010)). All the variants related to a functional component are included into an *adapt-manager*, which is deployed together the component in the same host. Then, when an adaptation is needed (i.e. an architectural variant must be introduced), the adapt-managers that are involved with this variant form an adapt-medium and collaboratively drive the adaptation process. Dynamic change is finally effected as the replacement of components by another variant, supported by a limited state transfer process. Low-level details about the safe stopping of involved elements, transactional support or propagation of changes to instances are not described. The approach is designed to support proactive (programmed) adaptations, although in a decentralized manner. More details about the distributed management of adaptations are addressed in (Zouari et al., 2010).

The main disadvantage of using decentralized approaches, or considering self-organised strategies, is that system-wide properties are more difficult to control. Each subsystem is able to evolve autonomously, but with the risk of breaking the system if no centralized control of the process is performed. Another disadvantage is that the maintenance of reconfiguration policies or architecture constraints is more complex, as a consequence of being scattered among the system. An interesting approach to manage the maintenance of reconfiguration policies could be the use of aspect-oriented approaches.

4.2.4 AOSD & Evolution Concerns

AOSD, i.e. Aspect-Oriented Software Development (Kiczales et al., 1997), proposes the separation of the crosscutting concerns of software systems into separate entities called aspects. This separation avoids the tangled concerns of software, allowing the reuse of the same aspect in different entities of the software system as well as its maintenance.

In the context of software evolution, several authors have claimed the importance of separating parts of the software that exhibit different rates of change (Mens & Wermelinger, 2002). This should be considered to avoid the entanglement of evolution concerns with other concerns, and to improve their design and maintainability. In this sense, aspects can be used to encapsulate the evolution concerns and avoid them to be scattered among the functional code.

The superimposition of aspects on software architectures has been shown as beneficial for the definition of multiple architectural views, the separate description of concrete concerns (such as dynamism), or the study of new composition schemas (Cuesta et al., 2006). For this reason, this section presents the proposals that have explicitly addressed the use of aspects in software architectures to encapsulate evolution concerns.

One of the earliest proposals on handling separately evolution concerns from functional concerns was the work of **Rasche and Polze** (Rasche & Polze, 2003). They implemented a framework on .NET supporting dynamic component reconfiguration. Configuration descriptions were described in a XML-based language, which provided constructs for defining simple ECA (Event-Condition-Action) rules. A centralized configuration manager evaluates configuration descriptions, initiates reconfigurations when required, and safely stops the components to be reconfigured. Only component instantiation and destruction was supported. One interesting contribution is that they used aspect-oriented programming to handle configuration code (i.e. the reconfiguration concern) separately from functional code. All the code required to support the reconfiguration process is provided to each component by means of a separate aspect. This aspect encapsulates transaction handling, connection management, start/initializing and attribute management. Thus, reconfiguration code is added transparently to binary components with a minimum interaction with the component developer.

Gustavsson et al. (Gustavsson et al., 2004) also propose to view the ability to update a system as an aspect, since this is a crosscutting feature. They define evolvability as a crosscutting concern in the sense that the mechanisms needed to make a software artefact dynamically updateable is orthogonal to the implementation of the artefact itself. They encapsulate the mechanisms for supporting dynamic updates of Java classes into the *runtime evolution aspect*, which is weaved to any Java class that may be updated at runtime. This evolution aspect works as a wrapper, which acts as a proxy to the users of the class. This aspect listens to a socket for update commands, which are sent through a graphical user interface. This approach supports multiple versions of classes, transference of state (by means of user-defined constructors) and safe locking of the affected classes and objects.

The previous approaches are very platform-dependent and focused on a medium granularity level (i.e. component implementation and classes, respectively). **SAFRAN** (David & Ledoux, 2006) is an extension of the Fractal component model (Bruneton et al., 2004) that uses AOP concepts and techniques to develop dynamic evolution code separately from business code. Whereas the previous approaches focused on encapsulating the *mechanisms* for dynamic evolution into an aspect, SAFRAN proposes to encapsulate reconfiguration *specifications* into aspects, called *adaptation aspects*. An adaptation aspect defines adaptation policies (mainly ECA rules), expressed in a domain-specific language for reconfiguration, called FScript (David & Ledoux, 2006), (David et al., 2009). FScript gives access to all the standard reconfiguration operations provided by Fractal, but it also allows the navigation and easy selection of elements. Adaptation aspects can be dynamically weaved or unweaved at runtime, thus allowing both reactive and proactive changes. The approach is very powerful and provides: (i) both internal & external event capturing, (ii) introspection capabilities to explore the running architecture and the properties of components, (iii) complete support for creating, destroying, binding and unbinding component instances, and (iv) consistency of reconfigurations through a transactional implementation, which allows the rollback of failed reconfigurations. This is supported by means of the logging of the inverse operations needed to undo each reconfiguration operation. The authors claim that all primitive Fractal reconfigurations are atomic and reversible, but they do not provide details about how removal operations are undone. The question is how the rollback of removed components and the restoration of their previous state are managed by means of only inverse operations. Another limitation of the approach is that the scope of adaptation is constrained to a composite Fractal component, so reconfigurations that may impact the entire architecture or may be synchronized with other composite components is not addressed.

This limitation is overcome by the **Fractal Aspect Component** model, FAC (Pessemier et al., 2008), an aspect-oriented extension of Fractal which supports the dynamic weaving of crosscutting concerns to multiple components. FAC uses the notion of *aspect component* to embody crosscutting concerns as components, and *aspect binding* to reflect the weaving among a component and an aspect component. The novelty introduced by FAC is the notion of *aspect domain*, which reflects the set of components that are affected by an aspect component, i.e. the impact of a crosscutting concern. The weaving interface provided in FAC is more general that the adaptation interface of SAFRAN, and thus, more expressive. Thus, by means of dynamic aspect weaving, FAC provides support for dynamic type evolution. However, it suffers from the same disadvantage of other aspect-oriented approaches: type evolution is only partially supported. Aspects are added to extend the functionality of the existing system, but the existing code cannot be removed (only intercepted to be hidden). FAC supports both proactive and reactive changes, because the weaving interfaces can be invoked either with the Fractal ADL or directly at runtime. In addition, since the weaving specifications can be defined outside the specifications of both aspects and components, FAC preserves the separation of the evolution concerns (encoded as weaving specifications).

Greenwood and Blair (Greenwood & Blair, 2006) combine several techniques to develop a flexible framework capable of adding self-adaptive behaviour to existing systems, i.e. without changing the underlying target system. Their work uses AspectWerkz (Vasseur, 2004) to dynamically weave aspects to existing code (by modifying the Java bytecode of the existing system), and Java-based reflection[10] to gather run-time information of the

[10] Java and .NET provide reflection mechanisms, which have been used by some proposals to gather run-time information about classes, methods, parameters and attributes. However, to be precise, these mechanisms only provide *introspection* mechanisms, but not *intercession* (which would enable the change of classes and attributes at runtime).

system. Self-adaptive behaviour is described by ECA-based policies, described in an XML file: the event is some joinpoint being reached in the target system, the condition is some test performed on the state of the target system, and the action involves the weaving or removing of an aspect. By contrast to other approaches, the result of an ECA rule is not a reconfiguration, but the weaving of an aspect. In their approach, they used Framed Aspects for specifying the set of changes that should be introduced in a running system. As an example, they weave two aspects for monitoring the system and effecting changes. However, as a result of using AOP techniques to adapt an existing system, only extensions to the target system can be done, but not the complete removal of components from the base code.

AspectLEDA (Navasa et al., 2007), (Navasa et al., 2009) is an extension of the formal ADL LEDA (Canal et al., 1999), which introduces primitives for describing Aspect-Oriented concepts. Aspects are modelled in LEDA as components, the join points that can be intercepted by aspects are the interactions among system components (i.e. public methods published by components), and the pointcuts that define the weavings of aspects to the base system architecture are modelled as special connectors. AspectLEDA provides support for dynamic adaptation (i.e. non-intrusive dynamic changes), both reactive and proactive. Reactive changes are supported through an external tool that allows the developer to dynamically weave new aspects to the base system. Proactive changes are defined in a structure called *Common Item*, which is produced together each aspect, and that defines ECA policies for deciding *when* an aspect must be applied, under *which* conditions or events, and *where* (i.e. which interaction point will be adapted by the aspect). Thus, AspectLEDA encapsulates the concerns related to dynamic adaptation into aspects, benefiting its reuse and maintenance. However, since only non-invasive changes are supported, major reconfigurations on the architecture cannot be performed, neither dynamic type evolutions on components.

AO-Plastik (Batista et al., 2008) is another approach that proposes to isolate reconfiguration concerns from functional concerns. AO-Plastik is an extension of the Plastik environment (Batista et al., 2005), a previous work of the same authors, which employs the AspectualACME ADL (Garcia et al., 2006) to support the modelling of crosscutting concerns on software architectures. AO-Plastik uses aspectual connectors –which represent crosscutting interactions- for encapsulating reconfiguration interactions, and aspectual components –which play a crosscutting role within an aspectual interaction- for encapsulating reconfiguration actions. In this way, the reconfiguration concern is separated into Events and Conditions (defined in aspectual connectors) and Actions (defined in aspectual components). This adds flexibility to the specification of dynamism: it allows that the same aspectual component (i.e. a set of reconfiguration actions) can act over different systems, according to different conditions defined in different aspectual connectors.

4.3 Comparison of the different approaches

This section analyses and compares the different approaches presented in the previous section. Although all the approaches address the dynamic evolution of software systems, they do from different perspectives (e.g. at a formal level, using aspects, implementing mechanisms for change, providing a framework, etc.), which make difficult their comparison. For this reason, a taxonomy[11] for dynamically evolvable systems is needed.

There are several taxonomies in the literature addressing the dynamic evolution of software systems: formal ADLs for self-management (Bradbury et al., 2004), mechanisms for dynamically composing software (Segal & Frieder, 1993) (McKinley et al., 2004a), dimensions of software change (Buckley et al., 2005), and self-adaptive systems (Huebscher & McCann, 2008) (Andersson et al., 2009a) (Salehie & Tahvildari, 2009). Among the different attributes proposed in these taxonomies, the following attributes have been selected: *degree of formality*, *activeness*, *evolution management*, *introspection*, and *type of changes*. Other attributes have been discarded because of being very specific (e.g. goal management, version management, etc.). In addition, other attributes not initially considered in existing taxonomies have been included in our comparison: *level of dynamism*, *separation of evolution concerns* (both specifications and mechanisms), and *consistency management*.

The following subsection describes these attributes in detail, and then, a comparison table is presented.

4.3.1 Description of the attributes selected

Next, we detail the attributes that have been used in our comparison and the reasons which motivated this selection.

- **Degree of formality**. This attribute defines if an approach is based on, or complemented by, any mathematical formalism. This is important to support the development of correct and robust dynamic software systems (Bradbury et al., 2004). Formal specifications have the following advantages: (i) they define the system at a high abstraction

[11] A system for naming and organizing things into groups which share similar qualities

level, (ii) they can be automatically analysed to verify properties or validate behaviour, and (iii) they can be used to generate code without ambiguity.

- **Level of dynamism**. This attribute refers to the kind of changes that can be done in a running system. *Dynamic reconfiguration* is supported if the system structure can be evolved at runtime, i.e. changes are performed at the architecture level. These changes have a coarse granularity: the smallest unit of change are components and/or connectors, which may be added or removed at runtime. In addition, these changes are done at the configuration level: they only impact a certain system instance, but not other instantiations. *Dynamic type evolution* is supported if types can be also defined or changed at runtime. That is, changes are not only structural, but also behavioural since the entire specification of an element can be changed at runtime. These changes generally have a medium or fine granularity: the smallest unit of change can be classes (medium granularity) or methods (fine granularity). Since changes are done at the type-level, they also impact all the instantiations of the changed type. Both kinds of dynamism are complementary and benefit each other (see section 3.6.3).
- **Activeness**. This attribute describes how changes can be initiated or driven: *reactively*, *proactively*, or *both*. Reactive changes are changes that are externally introduced at runtime, generally by means of a user interface or a tool. Proactive changes are changes that are initiated by the system, generally by means of the use of services for dynamic change. Both kinds of activeness are important and complementary: the support for proactive changes is essential to build self-management systems, whereas the support for reactive changes allows us to introduce unanticipated changes at runtime. For this reason, the level of activeness provided by an approach has been included in our comparison.
- **Separation of evolution concerns**. This attribute evaluates whether an approach separates evolution concerns from other concerns or not. This is important to improve the design and maintenance of both evolution code and functional code. In addition, it has been considered whether *evolution specifications* and *evolution mechanisms* are also separated. Evolution specifications define the proactive behaviour of a self-managed system: they define when and how the system will change at runtime. Evolution mechanisms are the underlying technologies and techniques that support runtime changes. Since functional concerns, evolution specifications and

evolution mechanisms have different rates of change, they should be kept separated to improve their maintenance.

- **Evolution management**. This attribute refers to the nature of the evolution management: if it is centralized or distributed. An approach has a *centralized evolution management* if dynamic change specifications and/or mechanisms are centralized in a specialized element or "configurator". Centralized evolution management is generally provided by approaches without proactive support, by means of an external tool. By the contrary, an approach has a *distributed evolution management* if dynamic change specifications are distributed across the elements of a system, i.e. different elements can perform dynamic changes on the running system. In some cases, however, the scope of changes that an element can perform is not detailed (e.g. Leda, Darwin). This attribute is important in order to account for the level of scalability provided by an approach.
- **Introspection**. This attribute describes the degree of self-awareness that a system is provided with. This is an essential attribute to build self-managed systems: a self-managed system, in order to reason and manage itself, first must be aware of itself. That is, it must be able to observe its internal state (i.e. its attributes and properties) and configuration (i.e. the elements is composed of and their relationships). This is generally provided by reflective approaches.
- **Types of change**. This attribute refers to the expressiveness for change that an approach supports. This is important to design highly flexible systems: the more expressiveness available, the more freedom for change provided. This expressiveness is limited by the dynamic evolution operations that are available for changing a running system. There are five types of evolution operations: (i) *additions*, if new elements (e.g. components, classes, methods) can be introduced at runtime; (ii) *removals*, if an element (e.g. a component or a class) can be removed from a running system; (iii) *updates*, if an element can be replaced at runtime; (iv) *linkings*, if the dynamic creation of links among elements is supported (e.g. connectors, relationships); and (v) *unlinkings*, if the removal of relationships among elements is supported. Depending on the number of evolution operations that are supported, this expressiveness has been defined as *Low* (1 or 2 evolution operations), *Medium* (3 or 4 operations) or *Full* (the 5 evolution operations are supported).
- **Consistency management**. This attribute evaluates the presence of mechanisms, or strategies, for preserving the consistency of a system

before and after a dynamic change. This is important to avoid that evolving a running system would result in its breakage. Three mechanisms have been considered: state transfer, safe stopping and transactional support. First, the *support for state transfer* guarantees that, in case of dynamic updatings, the previous state is preserved and is not lost among evolutions. Second, the *support for safe stopping* guarantees that the elements to be evolved are placed in a safe state. This avoids that changes on these elements may introduce inconsistencies in the running system. Third, *transactional support* allows a system to revert a set of dynamic changes if anything fails: e.g. execution of invalid actions, violation of system constraints, etc. This preserves system integrity after a dynamic change.

4.3.2 Comparison tables

A comparison table has been developed from the attributes and the approaches analysed in the above section. This table is divided in three separate tables due to the limitation of page size. These tables are grouped by approaches: formal approaches addressing dynamic reconfiguration (see *Table 1*), technological approaches (see *Table 2*), and self-adaptive and decentralized approaches (see *Table 3*).

The following symbols have been used:

✓ : An attribute is provided or supported.

✗ : An attribute has not been considered or is unsupported.

? : No information was available to evaluate an attribute.

½ : An attribute is partially supported

n.a. : An attribute is not applicable in this approach.

C | *D*: In the attribute *Evolution Management*, 'C' means *centralized* evolution management, whereas 'D' means *distributed* evolution management.

L | M | *F*: In the attribute T*ypes of Change*, this value refers to the level of expressiveness provided: Low 'L', Medium 'M', or Full 'F'.

			Level of Dynamism		Activeness		Separation of concerns					Consistency Management		
		Degree of Formality	*Reconfiguration*	*Type Evolution*	*Reactive*	*Proactive*	*Change Specs.*	*Change Mechs.*	*Evolition Management*	*Introspection*	*Types of Change*	*State Transfer*	*Safe Stopping*	*Transactional*
Process-Algebra	*Darwin*	✓	✓	×	×	✓	×	?	D	×	M	×	✓	✓
	LEDA	✓	✓	×	×	✓	×	?	D	×	F	×	×	×
	Dynamic Wright	✓	✓	×	×	✓	×	?	C	×	F	×	×	×
Graph-Based	*Hirsch et al.*	✓	✓	×	✓	×	×	?	C	×	M	×	×	×
	Wermelinger	✓	✓	×	✓	✓	✓	?	C	×	F	½	✓	×
	Baresi et al.	✓	✓	✓	✓	×	×	?	C	×	F	×	×	×
Reflective	*PiLar*	✓	✓	½[12]	×	✓	×	?	D	✓	F	×	×	×
	Archware ADL[13]	✓	✓	½[12]	✓	✓	✓	✓	D	✓	F	×	½[14]	×

Table 1. Comparison of formal approaches addressing Dynamic Reconfiguration

[12] Only partial support for type evolution is provided since changes are not propagated to all the instances of a type. Only local updates are performed.

[13] Considering the extensions of (Balasubramaniam et al, 2005).

[14] Only a weak stopping algorithm is implemented. Running instances are not safely stopped: their threads are only blocked when they try to interact via connectors. Since blocked threads have an internal transient state, this may leave the system in an inconsistent state for change.

			Level of Dynamism		Activeness		Separation of concerns					Consistency Management		
		Degree of Formality	*Reconfiguration*	*Type Evolution*	*Reactive*	*Proactive*	*Change Specs.*	*Change Mechs.*	*Evolition Management*	*Introspection*	*Types of Change*	*State Transfer*	*Safe Stopping*	*Transactional*
Procedural, OO	*Fabry*	✗	✗	✓	✓	✗	✗	✗	C	✗	M	✗	✗[15]	✗
	JDrums	✗	✗	✓	✓	✓	✗	✗	D	½	F	✓	✓[16]	✗
	Malabarba et al.	✓	✗	✓	✓	✗	✗	✗	C	✗	M	✓	✓[16]	✓
Dyn. Weaving	*EOS*	✗	✗	✓	✓	✓[17]	✗[18]	✓	C	✓	M	✗	n.a[19]	✓
	JAsCo	✗	✗	✓	✓	✓[17]	✓	✓	C	✓	M	✗	n.a[19]	✓
	PROSE	✗	✗	✓	✓	✓[17]	✗[18]	✓	C	✓	M	✗	n.a[19]	✓
Component-Based	*Simplex*	✗	✓	½[12]	✓	✗	✗	✓	C	✗	M	✓	✓	✓
	OpenCOM	✗	✓	½[12]	✗	✓	✗	✓	D	✓	F	✗	✓	✗
	Draco	✗	✓	✓	✓	✗	✗	✓	C	✗	F	✓	✓	✓
	Fractal (Julia)	✗	✓	½[12]	✗	✓	✗	✓	D	✓	M	✗	✗	✗
	SOFA 2.0	✓	✓	½[12]	✓	✓	✗	✓	D	½	M	✗	✗[15]	✗

Table 2. Comparison of technological approaches addressing Dynamic Evolution

[15] This approach has used a *passive partitioning* technique: old and new versions coexist at the same time (see section 3.4.2.1). Old versions are active until they become unreferenced by other elements. Thus, safe stopping criteria has not been used.

[16] Only a delayed stopping technique is implemented: changes are delayed until the element to change is inactive.

[17] In AOP approaches, proactive specifications are defined by weavings/pointcuts.

[18] To improve the separation of evolution concerns, weavings/pointcuts (as proactive specifications) should be separated from aspect code.

[19] Since change is performed directly at the level of code instructions (redirecting the execution flow to new code), safe stopping is not needed. When each thread reaches an interception point, it is redirected to a new location (i.e. the new code).

			Level of Dynamism		Activeness		Separation of concerns					Consistency Management		
		Degree of Formality	*Reconfiguration*	*Type Evolution*	*Reactive*	*Proactive*	*Change Specs.*	*Change Mechs.*	*Evolition Management*	*Introspection*	*Types of Change*	*State Transfer*	*Safe Stopping*	*Transactional*
Self-Adaptive	*Dashofy et al.*	✓	✓	×	✓	✓	✓	✓	C	✓	F	?	?	×
	Chisel	×	×	✓	✓	✓	✓	✓	C	✓	F	?	?	×
	Ramses	×	×	✓	×	✓	✓	✓	C	✓	F	×	×	×
	Rainbow	×	✓	×	×	✓	✓	✓	C	✓	F	×	×	×
	Plastik[20]	✓	✓	×	✓	✓	✓	✓	C	✓	F	×	✓	✓
	Ayed & Berbers	×	✓	×	×	✓	×	✓	C	×	F	✓	✓	×
Decentralized	*Georgiadis et al.*	✓	✓	×	×	✓	×	×	D	×	M	×	×	×
	K-Component v2	×	✓	×	×	✓	✓	✓	D	✓	F	✓	✓	×
	Evolver-Producer	✓	✓	½	✓	✓	✓	×	D	✓	F	✓	½[14]	×
	Adapt-Medium	×	✓	×	×	✓	✓	✓	D	×	F	×	×	×
AOSD-Based	*Rasche & Polze*	×	✓	×	×	✓	✓	✓	C	✓	M	×	✓	×
	Gustavsson et al.	×	×	✓	✓	×	×	✓	D	✓	F	✓	✓[16]	✓
	SAFRAN[20]	×	✓	×	✓	✓	✓	✓	D	✓	F	✓	×	✓
	FAC[20]	×	×	✓	✓	✓	✓	✓	D	✓	M	✓	×	✓
	Greenwood & Blair	×	×	✓	✓	✓	✓	✓	C	✓	M	×	n.a[19]	✓
	AspectLEDA	✓	✓	×	✓	✓	✓	?	D	✓	M	×	×	×
	AO-Plastik[20]	✓	✓	×	✓	✓	✓	✓	C	✓	F	×	✓	✓

Table 3. Comparison of self-adaptive and decentralized approaches

[20]Some of the attributes are inherited from the underlying middleware (i.e. OpenCOM or Fractal). These inherited attributes are shaded in grey.

4.4 Conclusions

After the analysis and comparison of different approaches supporting dynamic change in software architectures, it is possible to conclude that there is a gap among approaches addressing the description of dynamic changes at high levels of abstraction and technological approaches addressing the support for dynamic evolution. For instance, most of formal ADLs addressing architectural dynamism (e.g. Leda, Dynamic Wright, Baresi *et al.*, PiLaR) only focus on specifications, but not on the mechanisms that support this dynamism. As a consequence of this, formal ADLs do not provide primitives to allow the architect define how the old state should be transferred to new versions, or how an element should be stopped. On the other hand, technological approaches (e.g. JDrums, PROSE, Draco, Simplex) have focused on the feasibility of dynamic evolution mechanisms, but have omitted the support for proactive change behaviour, required to build self-managed systems. Advanced approaches, such as self-adaptive and decentralized ones, have focused on both the specification of dynamic changes and the mechanisms required for building self-managed systems. However, the support that is provided is very limited: consistency management is often incomplete, and dynamic type evolution is not addressed. In fact, any approach provides a complete support for both dynamic reconfiguration and dynamic type evolution. Only a few approaches provide a limited support (e.g. Draco, Julia, OpenCOM): at a low abstraction level, in a reactive way, and without the propagation of changes to all the instances.

As a result, it is necessary to provide a model that does not only describe both dynamic reconfiguration and type evolution at a high level of abstraction, but that also integrates the supporting mechanisms for dynamic change at the architectural level. This will enable the interaction of architectural elements with such mechanisms (e.g. a component may use reconfiguration services to evolve another component type), thus increasing the level of flexibility and expressiveness.

This model should include:

- A formal support for its specifications, to ease the automatisation of some tasks, such as the generation of code, the analysis and verification, etc.
- A high degree of expressiveness for specifying dynamic changes: different ways for initiating changes (i.e. proactively and reactively), introspection features, and a wide range of evolution operations available.

- An appropriate infrastructure to address different levels of dynamism: dynamic reconfiguration and type evolution.
- A separation of the evolution concerns, to increase their maintainability.
- A decentralized management of evolutions, to increase the scalability of evolvable systems.
- Mechanisms to guarantee the consistency of the architecture before and after dynamic changes.

This thesis presents a framework that has been defined to fulfil these needs. This framework is defined in PRISMA, a formal aspect-oriented ADL that has been extended with a set of primitives and mechanisms to support both the specification and management of dynamic changes at a high level of abstraction. This framework has been called *Dynamic Prisma*, and is presented as an important step towards developing highly flexible evolvable systems.

PART III

DYNAMIC PRISMA

Once the state of the art relative to dynamic software evolution has been introduced, this part of the thesis describes a framework to build architecture-based, dynamically evolvable, software systems. This framework has been called *Dynamic PRISMA*, because it has been applied on the PRISMA ADL, extending it to include dynamic evolution support.

The goal of this framework is to combine the two degrees of dynamism, i.e. Dynamic Reconfiguration and Dynamic Type Evolution, so that architecture-based systems can use both simultaneously. In some cases, a software system may need to reconfigure its structure to adapt to a concrete situation. In other cases, this system may need to change its architectural types at runtime, to correct errors or to add new behaviours. This dynamism is provided by following the principles from Autonomic Computing through asynchronous reflective mechanisms.

This part is structured in four chapters. First, Chapter 5 presents a case study from the domain of autonomous robotics which will be used to illustrate the concepts of Dynamic PRISMA. Next, Chapter 6 describes how dynamic reconfiguration is supported, illustrating how a composite instance can reconfigure its structure at runtime, either proactively or reactively. Then, Chapter 7 describes how dynamic type evolution is supported, illustrating how an architectural type can be modified at runtime and its instantiations automatically transformed to reflect the changes, but in an asynchronous way. Finally, Chapter 8 describes how the evolution semantics have been modelled by means of graph transformations.

CHAPTER V

CASE STUDY: AGROBOT

The purpose of this chapter is to provide an introduction to the case study that has been chosen to illustrate the Dynamic PRISMA approach. In addition, this chapter presents how dynamic evolution and reconfiguration can advantage the development of autonomous robotic systems, an area which could potentially benefit from the results of this thesis.

This chapter is organized as follows. First, section 5.1 introduces the field of agricultural robotics. Second, section 5.2 describes the specific requirements of robotic software architectures. Next, section 5.3 presents how agricultural robotics could advantage from dynamic evolution and reconfiguration. Section 5.4 presents Agrobot, the case study that is used to illustrate this work. Next section 5.5 presents the specific dynamic reconfiguration and evolution requirements of the Agrobot. Finally, section 5.6 presents the conclusions of this chapter.

5.1 Introduction: Agricultural Robotics

The agriculture is an outstanding sector in the Spanish economy. The mechanisation of some agricultural processes (e.g. seeding, weeding, harvesting), which are performed by large machines, has reduced considerably both the high labour and production costs. However, these machines generally are heavy weighted and big sized, which entail several disadvantages. First, the surroundings can be damaged due to the amount of space required to operate these large machines and to the soil compaction produced by their heavy weight. Second, the consumption of energy/combustible is high: both to move these large machines across the field and to transport them to the field. Third, they cannot be used in small fields (which are characteristic of Eastern Spain regions such as Valencia and Murcia): the narrow approach roads and borders of these small fields make very costly the transportation and use of large machines.

A friendly environmental solution, capable to operate in small fields, is the use of small agricultural robots (Blackmore et al., 2004) (Blackmore et al., 2006) (Pedersen et al., 2006). These agricultural robots are characterised by their small size and light weight, and by their ability to perform (some) autonomous tasks. A small size provides the robot with more manoeuvrability: (i) the robot can operate in smaller environments, such as small farms with narrow crop rows; and (ii) the robot can work at closest distances, looking after the plants more precisely. In addition, a small size (and a light weight) does not only help to reduce the manipulation and transport costs, but it also decreases the amount of energy/combustible the robot requires to work, thus providing the robot more operative autonomy.

A certain degree of autonomy makes the robot more productive and useful, since it does not require continuous human guidance to perform its tasks. The operator just introduces the robot in the target field, configures the tasks that the robot (or robots) must perform, and leaves the robot working. The robot will perform its tasks autonomously (i.e. without requiring human intervention) over long periods of time, until certain events occur: i.e. the tasks have been finished, detection of a particular disease, run out of resources, etc. Thus, a unique operator can manage several robots to take care of a big field or several smaller ones in parallel.

Figure 5.1 shows two designs of agricultural robots[21]. The Scamp scout (see Figure 5.1-(a)) is a concept vehicle for non contact sensing of growing crops. Valtra RoboTrac (see Figure 5.1-(b)) is a semi-autonomous tractor to perform pre-programmed tasks (such as tilling, plowing, disking, planting, etc.) that does not require human intervention. Note the small size of the robots with respect to current agricultural machines, which allows higher planting density and minimizes crop damage.

Figure 5.2 shows two real prototypes of agricultural robots. The prototype ACW (see Figure 5.2-(a)) is an autonomous Christmas tree weeder, which only cuts those weeds that are in competition with the trees and allows the non-competitive plants to improve biodiversity within the field. The prototype SAVAGE (see Figure 5.2-(b)) is a low cost, fully autonomous and highly flexible robotic platform specifically designed to perform agricultural tasks. It is based on a modular design to allow performing different agricultural tasks: the different equipment required can be accommodated easily over the base structure.

[21] http://www.unibots.com/Agricultural_Robot_Designs.htm

(a) Scamp scout[22]

(b) Valtra RoboTrac[23]

Figure 5.1. Agricultural robot designs

(a) ACW[24]

(b) SAVAGE[25]

Figure 5.2. Agricultural robot prototypes

5.2 Robotic Software Architectures

Given the nature of the Robotics domain, there are two qualities of special interest for the development of robotic software architectures: *dependability* and *adaptability*.

[22] Designed 2007 by Simon Blackmore and Massey Ferguson / AGCO
[23] Designed 2007 by Hannes Seeberg
[24] http://www.unibots.com/ACW.htm
[25] http://users.forthnet.gr/ath/startrek/

Robotic systems are integrations of a large number of sensor devices prone to malfunctions, such as wireless radio communications and vision sensors, complicating the need to coherently integrate the devices. The software systems operating robotic platforms must be capable of continued operation in the face of diminished hardware capacity and the loss of essential functions. Moreover, the information provided by hardware devices such as sensors may also exhibit a high degree of unreliability and intermittent spikes of erroneous sensor readings, necessitating the capacity to not only continue operating but also to compensate seamlessly for such errors. *Dependability* is an important feature in the design of autonomous robots, which must be able to deal with (partial) failures and continue working with reduced functionalities.

Another important goal is the need to operate within environments that can be *dynamic* and *unpredictable*. Developing a mobile robot that can traverse unknown terrain through varying weather conditions and with potentially moving obstacles, for example, greatly increases the difficulty of designing its software control systems in a way that can account for and continue operating under conditions not fully predicted during the system's design and development. *Adaptability* is an important feature to take into account in the design and development of the robots of the future, to enable them to survive in constantly changing environments.

However, dependability and adaptability are generally conflicting attributes: a robust system is specifically designed to tolerate predefined faults or situations, but is generally poorly adaptable.

5.3 Dynamic Evolution in Robotic Software Architectures

Dynamic Software Architectures can be used to support the adaptability and dependability attributes that a robotic system requires. A dynamic software architecture is one in which its elements (i.e. components and connectors) can be reconfigured or changed at runtime, without the need to shut the system down. These (dynamic) changes can be externally-driven (i.e. by the architect), or autonomously-driven (i.e. by the system itself) according to a set of (internal) high-level goals (Oreizy et al., 1999). Thus, a system is provided with enough flexibility to react and adapt, at runtime, to different events (predicted and unpredicted ones) from the environment. Dynamic changes may be related to the structure of a system or to its behaviour.

Dynamic reconfiguration (Magee & Kramer, 1996) is generally referred as the change that entails the modification of the architecture/structure of a system.

In robotic architectures, dynamic reconfiguration can be used to implement context awareness behaviours (i.e. adaptability behaviours): adjust current strategies to external conditions (e.g. weather conditions, unavailability of resources, blocked paths …) or to internal conditions (e.g. low level of energy). It can also be used to implement fault-tolerance behaviours (e.g. removal or replacement of failed components, optimization in case of reduced functionality …).

Dynamic type evolution or *dynamic updating* (Segal & Frieder, 1993) is generally referred as a change that entails the modification of the behaviour of the components. In robotic architectures, dynamic type evolution may be useful to support unforeseen changes, like the addition of new behaviours at runtime (e.g. addition of new tools and capabilities to a robot), or to update malfunctioning components.

Dynamic reconfiguration and type evolution are two important features to take into account in the design and development of the robots of the future, to enable them to survive in constantly changing environments. Currently, these dynamic changes are provided at low-abstraction levels, i.e. at the code level (attributes, methods and code modules). This is poorly reusable and maintainable: it relies on current implementation structures and its mechanisms are highly scattered around the software architecture. This is not acceptable: the increasing demand of service robots (IFR, 2009) is pressuring engineers to reduce their development cost and time-to-market, as well as requiring better adaptability, dependability and overall quality. Better high-level designs and developments are needed (Schlegel et al., 2009).

The approach presented in this thesis copes with these needs. Dynamic reconfiguration and type evolution are provided: (i) as independent concerns, which facilitates its reuse and maintenance; and (ii) at a high abstraction level, i.e. the architecture level: changes are performed on higher level artefacts, such as components and connections, instead of at the code level.

5.4 Case Study: Agrobot, An Autonomous Robot for Plague control

To illustrate the approach, in this section it is presented the software architecture of Agrobot, an autonomous agricultural robot for plague control. Its objective is to patrol -at periodical intervals- a small field or delimited area, looking for pests or disease attacks over a set of growing crops. When a threat is detected, a pesticide is applied to the field, as a first counter-attack measure, and a real-time alarm is sent to the manager, to take further specialized actions.

5.4.1 Main Architecture

The Agrobot architecture is defined hierarchically as a system of systems, that is, a composition of composite components. The top level, shown in Figure 5.3, describes the set of composites the robot is composed of and their communication channels (see Figure 5.4 for more details about the notation used). Each composite component is depicted as a component, which provides and requires a set of services through its ports. The interaction among composites is coordinated by different connectors (represented as blue small components).

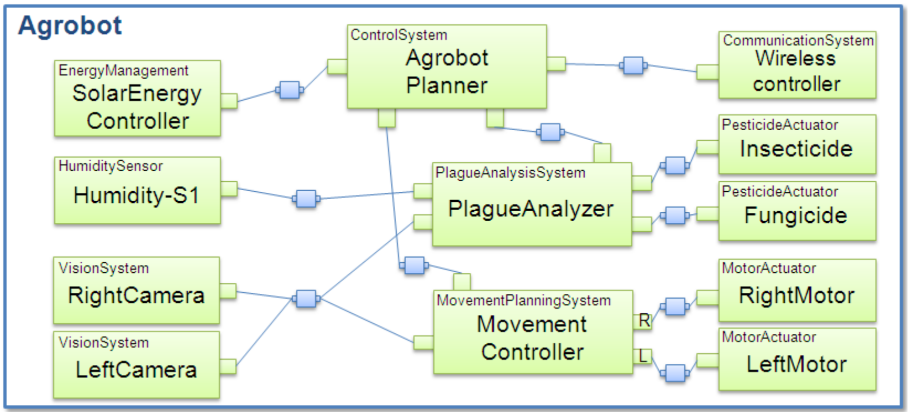


Figure 5.3. Software architecture of the Agrobot

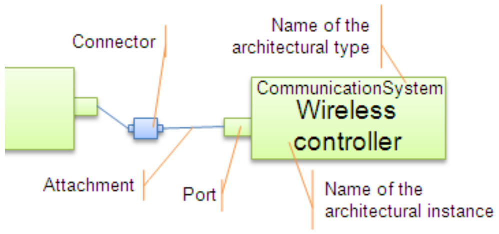

Connector
Name of the architectural type
CommunicationSystem
Wireless controller
Attachment
Port
Name of the architectural instance

Figure 5.4. Notation used

More specifically, these composites are instances: local deployments of component types. Component types are reusable artefacts which define the structure and behaviour of a software artefact. Component instances are the execution of a type in a specific environment (i.e. the Agrobot architecture), which are conveniently configured or parameterized. Each component instance not only depicts the instance name (e.g. LeftCamera, see Figure 5.3, bottom-left), but also the name of its component type (e.g. VisionSystem).

The Agrobot is provided with a set of high-level behaviours or tasks: analyse a growing crop, apply a pesticide, send an alarm, move to the next crop, recharge energy, or sleep. The decision of which specific behaviour should be carried out is performed by a composite called *AgrobotPlanner* (see Figure 5.3). This composite coordinates and plans the set of actions that must be done to fulfil the selected behaviour: image capturing, pattern analysis, movement, sensing, communication, pesticide activation, element disconnection, etc. These actions are provided by the different composites of the Agrobot architecture, which are parameterized or configured to carry out each specific action.

Next, each one of the composite components of the Agrobot architecture shown in Figure 5.3 is briefly described. Vision is performed by the *RightCamera* and *LeftCamera* composites. These components capture and pre-filter real-time images from the environment. These images may be used by other components to look for crop diseases (i.e. the *PlagueAnalyzer* component) or to guide the movement (i.e. the *MovementController* component). Sensing is provided by the *Humidity-S1* component, which manipulates a set of sensors to obtain the humidity of the environment and the soil. This allows the *PlagueAnalyzer* component to evaluate the health of the crop being observed. Energy management is provided by the *SolarEnergyController* component, which recharges the batteries and controls its consumption. When the battery levels are low and the current position does not allow recharging the batteries, this component alerts the *AgrobotPlanner* component to look for a better position (i.e. with more light). Movement planning and control is provided by the *MovementController* component. It takes the images captured from its environment (by the *RightCamera* and *LeftCamera* components) to generate the sequence of movements to go towards the next target, avoiding obstacles. Each movement is carried out by the motor actuators: the *LeftMotor* and *RightMotor* components, which control left and right wheels respectively. Plague analysis and reaction is performed by the *PlagueAnalyzer* component. It contains information about the plagues and diseases to look for in the crops, as well as the actions to perform for each threat detected. This component analyzes the images captured by the vision components (i.e. *RightCamera* and *LeftCamera*) looking for potential threats. When a threat is detected, first an alarm is sent to the manager (through the *WirelessController* component). Next, depending on the degree of the threat, the Agrobot applies a small quantity of pesticides (managed by the *Insecticide* and *Fungicide* components) directly to the damaged area (thus avoiding the overuse of pesticides).

Next, among the different composites of the Agrobot, we will focus on details of the Vision subsystem. The goal is to provide an idea of how the different elements have been specified by means of the PRISMA ADL. For this reason, only an artefact of each kind of PRISMA element is described: a composite component type (i.e. a System) and its instantiation (i.e. a Configuration), a simple component type (i.e. a Component), and an Aspect. Further details about the other elements of the VisionSystem are provided in Appendix A, section A.1 (see page 381). This will permit to illustrate later how the different kinds of PRISMA elements (i.e. Systems, Configurations, Components and Aspects) can be dynamically evolved.

5.4.2 Composite Components: VisionSystem

Most of the components of the Agrobot architecture are composite components: each one is composed of other components and connectors. The structure and behaviour of composite components are specified by means of PRISMA Systems (see section 2.4, the PRISMA model). A PRISMA System defines a generic structure, or pattern, which can be reused in any software architecture where necessary. This pattern specifies:

1. What are the architectural types (components and connectors) that are valid in the System instances,
2. How many instances of these types can be created,
3. How these types can be connected to each other, and
4. Which are the provided and required behaviours.

For instance, Figure 5.5 depicts the pattern described by the *VisionSystem* architectural type: (i) it is composed of two component types, *VideoCaptureCard* and *ImageProcCard*, and two connectors, *VCC-Conn* and *IPC-Conn*; (ii) these types are singleton, i.e. only one instance can be created of each one; (iii) these types mainly implement a pipe-and-filter style (Taylor et al., 2009); and (iv) the results of internal processing are published through a port called *ImgOutputPort*.

The *VideoCaptureCard* component type encapsulates a hardware device which captures images from the environment at a constant frame rate. The *ImageProcCard* component type encapsulates another hardware device which pre-processes the images captured. This component provides specific hardware-based video processing algorithms, which are optimized for the robot needs. The *VCC-Conn* connector forwards the captured images of the *VideoCaptureCard* component to the *ImageProcCard* component. The *IPC-Conn* connector forwards the processed images to the *ImgOutputPort* port, which are then made available to other subsystems of the Agrobot.

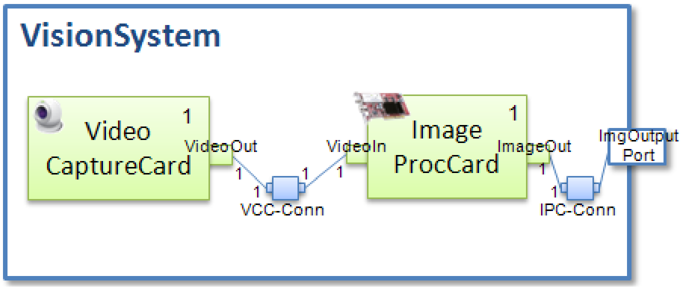


Figure 5.5. VisionSystem architectural type

The PRISMA specification of the *VisionSystem* type is provided in Figure 5.6. This specification defines a System called *VisionSystem* and provides additional details that are omitted from the graphical depiction. For instance: the name of the attachments and their cardinalities, the maximum and minimum cardinalities of architectural elements, or the constructor and destructor of the type.

```
System VisionSystem

   Ports
      ImgOutputPort : I_ImageProcessingServices;
   End_Ports;

   Import Architectural Elements
      VideoCaptureCard(1,1), ImageProcCard(1,1), VCC-Conn(1,1),
               IPC-Conn(1,1);

   Attachments
      Att_VCC_VCCConn: VideoCaptureCard.VideoOut(1,1) <-->
            VCC-Conn.VideoIn(1,1);
      Att_VCCConn_IPC: VCC-Conn.VideoOut(1,1) <-->
            ImageProcCard.VideoIn(1,1);
      Att_IPC_IPCConn: ImageProcCard.ImageOut(1,1) <-->
            IPC-Conn.ImageIn(1,1);
   End_Attachments;

   Bindings
      Bin_IPCConn: ImgOutputPort(1,1) <--> IPC-Conn.ImageOut(1,1);
   End_Bindings;

   /* Constructor definition */
   new ( frameRate: integer; cameraPosition: string)
   {
      new VideoCaptureCard(frameRate);
      new ImageProcCard(cameraPosition);
      new VCC-Conn();
      new IPC-Conn();
      new Att_VCC_VCCConn(input VideoCaptureCardID: string,
            input VCC-ConnID: string);
      new Att_VCCConn_IPC(input ImageProcCardID: string,
            input VCC-ConnID: string);
      new Att_IPC_IPCConn(input ImageProcCardID: string,
            input IPC-ConnID: string);

      new Bin_IPCConn(input ImageProcCardID: string);
   }

   /* Destructor definition */
   destroy()
   {
      destroy VideoCaptureCard(frameRate);
      destroy ImageProcCard();
      destroy VCC-Conn();
      destroy IPC-Conn();

      destroy Att_VCC_VCCConn();
      destroy Att_VCCConn_IPC();
      destroy Att_IPC_IPCConn();

      destroy Bin_IPCConn();
   }

End_System VisionSystem;
```

Figure 5.6. PRISMA specification of the VisionSystem type

In PRISMA, the instantiation of a System is defined at the configuration-level, and is called *Configuration* (see section 2.4). Thus, the PRISMA specification for instantiating and configuring these instances is provided in Figure 5.7. This specification defines two Configurations called *RightCamera* and *LeftCamera*; it instantiates each of the architectural types defined in the System type and connects them appropriately. The configurations *RightCamera* and *LeftCamera* mainly differ in that they are *parameterized* to use a right camera or a left camera, respectively. That is, they differ in that they have a different state and are differently connected.

```
Architectural_Model_Configuration RightCamera =
   new VisionSystem {
      Right-VCapt = new VideoCaptureCard(30);
      ImgProc-1 = new ImageProcCard("right");
      VCC-Conn1 = new VCC-Conn();
      IPC-Conn1 = new IPC-Conn();

      att1 = new Att_VCC_VCCConn(Right-VCapt, VCC-Conn1);
      att2 = new Att_VCCConn_IPC(ImgProc-1,  VCC-Conn1);
      att3 = new Att_IPC_IPCConn(ImgProc-1, IPC-Conn1);

      bin1 = new Bin_IPCConn(ImgProc-1);
   }
```

```
Architectural_Model_Configuration LeftCamera =
   new VisionSystem {
      Left-VCapt = new VideoCaptureCard(30);
      ImgProc-2 = new ImageProcCard("left");
      VCC-Conn2 = new VCC-Conn();
      IPC-Conn2 = new IPC-Conn();

      att1 = new Att_VCC_VCCConn(Left-VCapt, VCC-Conn2);
      att2 = new Att_VCCConn_IPC(ImgProc-2, VCC-Conn2);
      att3 = new Att_IPC_IPCConn(ImgProc-2, IPC-Conn2);

      bin1 = new Bin_IPCConn(ImgProc-2);
   }
```

Figure 5.7. PRISMA specification of RightCamera and LeftCamera instances

For instance, Figure 5.8 depicts graphically the internal structure (i.e. the architecture) of the *RightCamera* composite component instance.

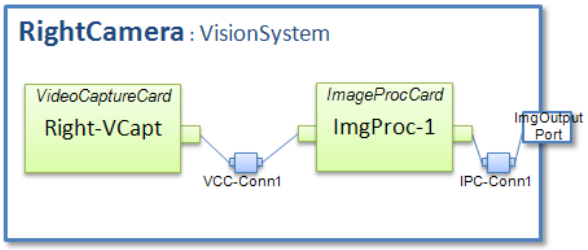


Figure 5.8. White-box view of the RightCamera component instance

5.4.3 Simple Components: ImageProcCard

As an example of how simple components are defined in PRISMA (and which allows us later to illustrate how these can be evolved), this section shows the specification of the *ImageProcCard* component. This component encapsulates a hardware device for the pre-processing of captured images. The PRISMA specification of this component is provided in Figure 5.9.

ImageProcCard is a PRISMA component that is *invasively composed* (see section 2.4.3, page 51) of two aspects: an Integration aspect, *ImageProcCardController*, and a Presentation aspect, *ImageProcCardGUI*. The Integration aspect is a kind of aspect which allows the integration of external libraries or components (such as COTS, i.e. components off the shelf) into PRISMA specifications (Pérez et al., 2008). This aspect provides services which control the image processing hardware device.

On the other hand, the Presentation aspect is a kind of aspect which captures user interface concerns. This aspect provides a visual interface used mainly for debugging purposes: it shows the input and output images, and some statistic results.

The *ImageProcCard* component provides two ports: *VideoIn*, which receives the captured images, and *ImageOut*, which outputs the processed images. These ports are bound to the VIDEOCARD and IMAGEANALYZER roles of the *ImageProcCardController* (see played_roles, ports section, Figure 5.9). This means that the services of these ports are actually provided/required by this aspect.

```
Component ImageProcCard

   Integration Aspect import ImageProcCardController;
   Presentation Aspect import ImageProcCardGUI;

   Ports
      VideoIn  : I_VideoServices,
         Played_Role ImageProcCardController.VIDEOCARD;
      ImageOut : I_ImageProcessingServices,
         Played_Role ImageProcCardController.IMAGEANALYZER;
   End_Ports

   Weavings
      ImageProcCardGUI.showImage(image)
         after
      ImageProcCardController.newProcessedImage(image);
   End_Weavings

   new(cameraPosition: string) {
      ImageProcCardController.begin(cameraPosition);
      ImageProcCardGUI.begin();
   }

   destroy() {
      ImageProcCardGUI.end();
      ImageProcCardController.end();
   }

End_Component ImageProcCard;
```

Figure 5.9. PRISMA specification of the ImageProcCard component

However, the behaviour of PRISMA components is not provided by a single aspect, but by the *weaving* (or gluing) of several aspects. In this case, the behaviour of *ImageProcCard* results from the weaving of the integration and presentation aspects (see weavings section in Figure 5.9): each time a new image is processed (i.e. the service *newProcessedImage()* is executed), the service *showImage* is executed afterwards.

Finally, the component specification also describes how the component is instantiated and destroyed (see the *new* and *destroy* services).

5.4.4 Aspects: ImageProcSwController

Finally, in order to illustrate how the behaviour and state of aspects is defined, the *ImageProcSwController* aspect is shown.

ImageProcSwController is a Functional Aspect which provides a software-based version of the image processing algorithms. This aspect is provided as an alternative implementation in case the hardware device for processing images fails.

The internal state of the *ImageProcSwController* aspect consists of two attributes (see the attributes section, Figure 5.10): the image that has been captured, *CapturedImage*, and the last image that has been processed, *LastProcessedImage*. Both attributes are variable, i.e. their value can be changed at runtime. *Image* is a user-defined data type which mainly has two values (see section A.1.2 in page 382): the ID of the image and a stream of bytes with the image contents.

The services that the *ImageProcSwController* aspect provides and requires are defined in the *Services* section. There are four kinds of services:

1. Initialization (*begin*) and finalization (*end*) services. These services define how the aspect is initialized or finalized, respectively. In this example, the begin service initializes the constant attribute *spatialPosition.*
2. Input services (defined with the "in" keyword), which define provided behaviour. The *newCapturedImage* service is invoked to process a new image.
3. Output services (defined with the "out" keyword), which define required behaviour. The *newProcessedImage* service is invoked by the aspect when a new processed image is available (i.e. ready to be analysed by other subsystems of the robot).
4. Private services, which define internal behaviour. The *processImage* performs the processing of an image.

The behaviour of each service is defined in a subsection called *valuations*, which uses a semantics based on modal logic of actions (Stirling, 1992).

```
Functional Aspect ImageProcSwController
   using I_VideoServices, I_ImageProcessingServices

   Attributes
      Constant
         spatialPosition : string;
      Variable
         CapturedImage : Image;
         LastProcessedImage : Image;
   Services
      // Initialization service
      begin(input cameraPosition: string);
         Valuations
            (cameraPosition==“right”) or (cameraPosition==“left”)
               [begin(cameraPosition)]
            spatialPosition = cameraPosition;

      // New captured image notification
      in newCapturedImage(input capturedImage: Image);
         Valuations
            [newCapturedImage(capturedImage)]
            CapturedImage = capturedImage;

      // Notifies about the output of a new processed image
      out newProcessedImage(input processedImage : Image);
         Valuations
            [newProcessedImage(processedImage)]
            processedImage = LastProcessedImage;

      // Image processing service
      processImage();
         Valuations
            [processImage()]
            LastProcessedImage = ImageProcessing(CapturedImage);

      // Finalization service
      end();

   PlayedRoles
      VIDEOCARD for I_VideoServices ::=
         newCapturedImage?(capturedImage);
      IMAGEANALYZER for I_ImageProcessingServices ::=
         newProcessedImage!(processedImage);

   Protocol
      IMAGEPROCESSINGCARDCONTROLLER ::=  begin():1 --> CAPTURE;
      CAPTURE ::= end():10  +
         (  VIDEOCARD_newCapturedImage?(capturedImage):10 -->
            processImage():10 -->
            IMAGEANALYZER_newProcessedImage!(processedImage)):10
         ) --> CAPTURE;

End Functional Aspect ImageProcSwController;
```

Figure 5.10. PRISMA specification of the ImagProcSwController aspect

The *Played_Roles* section defines, using a π-calculus semantics (Milner, 1993), the services that are required and provided through each *played_role*. In PRISMA, a *played_role* defines a point of interaction, or a hook, of the aspect. This hook is later bound to a port when the aspect is imported by a component. In this way, aspects are defined independently of components, and are able to publish and constrain their behaviour through component ports.

Finally the *protocols* section, using a π-calculus semantics extended with priorities (Perez, 2006), defines the different states of the aspect and how the different services are orchestrated. In this example, the aspect has two main states:

- `IMAGEPROCESSINGCARDCONTROLLER` state, i.e. the initial state.
- `CAPTURE` state, on which images are processed this way: an image is received through the *VIDEOCARD* played_role, it is processed, and the result is sent through the *IMAGEANALYSIS* played_role.

Note how for each service invocation (postfixed with the symbol "!") or request (postfixed with the symbol "?"), its execution priority and the new state is described.

5.5 Dynamic Evolution Requirements of the Agrobot

The Agrobot system requires dynamic reconfiguration and evolution features to support functional and non-functional requirements. Two examples of functional and non-functional requirements that take advantage from dynamic reconfiguration support are energy-management and fault-tolerance, respectively.

On the one hand, energy management is a functional requirement that optimizes the energy usage of the robot. When energy levels are low, the architecture should be automatically reorganised to disable or remove those components that are non-critical (e.g. humidity sensors, pesticide actuators, plague analysis, etc.). Only those components concerned with movement planning and energy management should be kept active, in order to reach a recharging location and to save as much energy as possible. And the removal of non-critical components must be done safely, to avoid leaving the system in an undesirable state: if a plague analysis was being performed, this analysis must be stopped in a way that after the reactivation it continues in the same point it was stopped (i.e. in the same crop). This is supported by means of dynamic reconfiguration.

On the other hand, fault-tolerance is a non-functional requirement that provides reliability to the system: in case any component fails, the robot must be aware of it and react accordingly. For instance, if one of the cameras is malfunctioning (e.g. RightCamera), it should be excluded from image analysis; then, only the other spare camera (e.g. LeftCamera) should be used. And, if alternative components are available, the robot must be able to dynamically replace the malfunctioning component by an alternative one.

Next, to illustrate and evaluate the approach presented in this thesis, two scenarios are presented: one that uses dynamic reconfiguration (see section 5.5.1) and another that uses dynamic type evolution (see section 5.5.2).

5.5.1 Dynamic reconfiguration scenario: VisionSystem fault-tolerance support

The artificial vision system plays an important role for the autonomous execution of tasks, such collision detection or plague detection. For this reason, this subsystem is one of the critical elements of the Agrobot: if it does not work properly, the robot will not be able to perform its tasks or even survive (e.g. if energy is very low, then it could not locate a recharging location). Therefore, a certain degree of fault-tolerance must be supported.

Fault-tolerance support through Dynamic Reconfiguration

There are several design strategies and architectural patterns for supporting fault tolerance (Heimerdinger & Weinstock, 1992). These strategies generally rely on the use of redundancy: each critical element of the system has several replicas concurrently executing in the system. Thus, the failure of one of these critical elements is *masked* by the other running replicas, being the system able to continue working transparently. The advantage of using this kind of redundancy is that recovering time is immediate, since the different replicas are running and have the same state that the faulty element. However, this approach has several disadvantages:

- Synchronization points, where the results of the different replicas are compared or merged, are difficult to define and implement;
- Since each replica is concurrently running, this increases the usage of system resources and costs;
- System complexity increases proportionally to the number of replicas, due to the number of messages each replica produces;
- It is a static approach: it cannot react to runtime changes of the system.

An alternative to (online) redundancy for supporting fault-tolerance is the use of dynamic reconfiguration (Yurcik & Doss, 2001). Dynamic reconfiguration provides a software system with flexibility to change its configuration and/or replace its components at runtime. Thus, in case of a failure, the system can reconfigure its elements at runtime to use a different configuration (i.e. using different alternative elements) or reach a safe state. This does not require that the alternative elements or configurations are executing: they may be loaded dynamically on demand.

Thus, the usage of dynamic reconfiguration in the design of fault-tolerance systems would avoid not only the complexity of keeping several redundant components operative (since synchronization points are not required), but also its related costs (since alternative elements can be dynamically loaded at runtime). In addition, dynamic reconfiguration is dynamic itself: it can be used to react at runtime to new, non-predicted situations.

Fault-tolerance in the VisionSystem type

Dynamic reconfiguration is used in the *VisionSystem* elements (i.e. the *RightCamera* and *LeftCamera* components) to change their internal architecture when a fault is detected in one of its (internal) components. Fault detection is performed by a watchdog component, *VisionWatchdog*, which periodically checks if images are being captured and processed correctly. In case misbehaviour is detected, this component sends an event (i.e. raises an exception) to notify a failure. For instance, if the image processing component (*ImageProcCard*) does not correctly process images or has a negative performance, then the *VisionWatchdog* component sends the following event:

```
faultyOutput!(out `ImageProcCard`)
```

A similar event is raised if the *VideoCaptureCard* component does not provide images or are provided with a very low quality (e.g. almost black):

```
faultyOutput!(out `VideoCaptureCard`)
```

The events triggered by the watchdog component should be captured by self-management mechanisms providing dynamic reconfiguration, and react appropriately for each kind of event. There are two kinds of reactions that can be performed: a replacement by another element, or the disablement and isolation of the failing component in order to avoid error propagation.

A failed component can be dynamically replaced if alternative components are available (either spare hardware devices that are initially switched off, or alternative software-based implementations that are initially unloaded). This is the case of the *ImageProcessingCard* component. In case a failure is detected, it must be removed (i.e. the hardware device is deactivated) and another, alternative, component must be used instead: a component called *ImageProcSoftware*. This component implements another (compatible) image processing algorithm, but with less performance than the removed one. The reason is that hardware-based algorithms are generally faster than software-based algorithms. For this reason, to make up for performance, several instances of the *ImageProcSoftware* are created, so that the overall performance would be similar to the provided by the *ImageProcCard* component.

If no alternative components are available, then the failure must be managed in order to avoid its propagation to the other elements of the system. This is the case of the *VideoCaptureCard* component. In case it fails, the entire *VisionSystem* instance must be disabled, since it cannot provide input data. Then, the other elements of the Agrobot must be aware of this and avoid its usage. Since the Agrobot is provided with two *VisionSystem* instances, *RightCamera* and *LeftCamera*, in case one of them fails, the other one must be used exclusively. This situation is masked by the connector that propagates the images from the cameras (see Figure 5.3).

Then, the reconfiguration process should do the following upon receving a notification of the *VisionWatchdog* component:

If the *ImageProcCard* instance is failing:

1. This instance should be disabled and removed from the architecture. This implies that all its connections must be also removed.
2. New instances of *ImageProcSoftware* should be created and attached to the elements that the failing *ImageProcCard* instance was attached.

If the *VideoCaptureCard* instance is failing:

1. Another event should be propagated to the outside, to notify about this situation: a *VisionSystem* instance cannot behave as expected, so the other elements must adapt to the new context.
2. Internally, the VisionSystem instance should disable its elements and set a timer to periodically check if the failure is still present. The reason is that the failure may be temporal or recover after a self-restarting process.
3. Externally, the Agrobot should remove the attachment from the failing *VisionSystem* instance to the connector, and notify the connector about this. Thus, the connector will not interpolate two input images, but only one.

It is important to take into account that the rest of elements that interact with a *VisionSystem* instance (e.g. *PlagueAnalyzer* or *MovementController*) must not be aware of the reconfiguration process that is performed internally.

The architectural type of the VisionSystem, extended with the elements required for fault-tolerance described above, is depicted in Figure 5.11. Note how the pattern defined by the PRISMA architectural type allows us to describe alternative configurations: the cardinality of *ImageProcSoftware* (0..*)

shows that it can be instantiated or not. In addition, this cardinality also shows that several instances of this component can be created. This is also shown by the cardinality of the allowed connections (depicted as '*'): an instance of *VCC-Conn* can be attached to several instances of *ImageProcSoftware*. An example of an alternative configuration is depicted in Figure 5.12: it depicts the configuration of the *LeftCamera* component in case a failure in the *ImageProcCard* instance would be detected.

Note also that the *FaultyOutput* port of the *VisionWatchdog* component remains disconnected: the reason is that this port is used to raise events/exceptions, which may be captured or not. In this case, these events are captured by the reconfiguration mechanisms, as it will be described in Chapter 6. For further details, the complete specification of the VisionSystem and its components (including the watchdog and alternative components) is provided in appendix A.1 (see page 381).

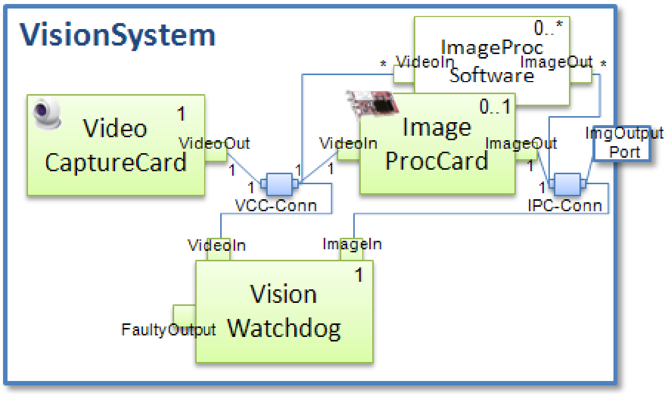


Figure 5.11. VisionSystem type with fault-tolerance support

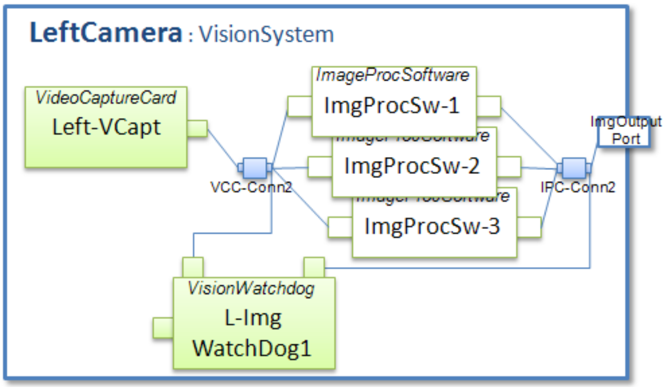


Figure 5.12. Example of alternative configuration in case of failure

Only two possible failures that may occur in the context of a *VisionSystem* instance have been described. However, the reader must take into account that there is a broad range of situations that must be also covered to provide an acceptable fault-tolerance degree, such as: the failure of connectors, the failure of the watchdog component itself, or even the failure of the reconfiguration mechanisms.

5.5.2 Dynamic type evolution scenario: Changing the ImageProcSoftware component

As shown in the previous section, dynamic reconfiguration can be used to provide a large complex system with self-management properties, such as for supporting fault-tolerance. However, self-management has some limitations to take into account: it can only reconfigure or adapt a system according to a range of foreseen situations and/or by using a previously available set of software components or types.

What happens then if unforeseen changes are required, such as maintenance operations, the introduction of new features, or even the modification of the self-management subsystems? For instance, in the case of the Agrobot, it may require updates to correct bugs or the addition of new components to support new tools or behaviours. These kinds of changes concern the types[26] that define a system, i.e. its building elements, which are outside the scope of dynamic reconfiguration.

Convenience of Dynamic Type Evolution

Changes concerning types (i.e. the specification of software artefacts) are generally performed offline: they are applied on the source code of the target system, and are installed after the rebooting of the target system. However, this solution is unfeasible for systems with high-availability requirements, which cannot stop their activity under any reason. Examples of such systems are those that are in charge of critical infrastructures (e.g. related to military resources, energy, health or transports), those required to operate 24 hours a day 7 days a week (e.g. banking systems, manufacturing industry systems), or those that are not reachable (e.g. autonomous robots in space explorations). In such systems, changes or maintenance operations must be performed at runtime, while the system keeps operative. It is in this context where *dynamic*

[26] A type, in this context, is the specification of the structure and behaviour of a software artefact, together with its realization (i.e. the executable code). For more details, see section 7.2.1-“Definitions of Type, Instance and Architectural Type” in page 248.

type evolution is very convenient, or even a necessity, to deal with the expected longevity of large and expensive systems.

This is also the case for the Agrobot: it is an autonomous robot which must be fully operative, day and night. During the day it monitors weather conditions, supervises the growing crops looking for potential threats, or recharges itself. During the night, it processes the data collected during the day, keeps active surveillance looking for intruders (e.g. animals that are harmful for the crops), and continues monitoring the weather conditions. Some of these tasks are critical and cannot be interrupted under any reason. For instance, the energy management task is responsible for saving enough energy to perform the next recharge cycle; the weather monitoring task is responsible for keeping the robot safe from adverse conditions; and the surveillance task is responsible for reacting quickly to potential threats[27].

In case updates or new functionalities may be required, they should be applied in the Agrobot without stopping or rebooting it, because such an operation would disrupt the above described tasks. For instance, if the energy is low, a complete shutdown may consume the energy required for the next recharging cycle. Another example: during a rebooting process, the weather conditions may change so quickly that the safety of the robot or the monitored crops may be seriously exposed to danger. For these reasons, to cope with the maintenance and/or extensibility needs of the Agrobot whereas keeping it operative, *dynamic type evolution* has been introduced.

Dynamic type evolution is used in the following two scenarios: *software updating*, or *hardware-software updating*. *Software updating* is performed to remove bugs from existing components or to introduce components with new functionalities. In this case, there is no direct user intervention: the Agrobot receives the updates through the *Communication System* (see the component *WirelessController* in Figure 5.3, page 140) and, by means of the dynamic type evolution mechanisms (which are described in Chapter 7), they are installed automatically whithout stopping the entire system.

Hardware-software updating is performed to replace a hardware device by another or to add a new tool (e.g. new pesticides, new sensors, different motors, etc.). This is an operation which also needs to update the software architecture in order to be integrated with the other elements. In this case, the

[27] Certain threats can be only addressed in their earlier states. Otherwise, the threat propagates quickly and it will not be able to be treated, thus causing the crop to be lost. In case one of such diseases is detected, the Agrobot must react quickly with the appropriate measures (i.e. the use of specialized pesticides, the removal of affected crops, or the sending of an alert to the farmer).

only direct user intervention that is required is the plugging of the new hardware device in the Agrobot. Once the device is plugged in the Agrobot, it provides the Agrobot with the software updates required, which are installed in the same way as in the software updating case described above.

In both cases, a software updating artefact is used: it contains the new type(s) to introduce in the system and the instructions to introduce this(these) type(s) into the running system seamlessly. In fact, as it will be described in Chapter 7, these instructions are actually a set of *dynamic evolution operations*, which allows us to change the system at runtime without shutting it down and without user interaction. In this way, the users of the Agrobot (i.e. the farmers) are provided with remote maintenance support and extensibility, without disrupting the critical tasks of the robot.

Dynamic updating in the VisionSystem type

In order to later illustrate how our proposal supports dynamic type evolution, next a software updating scenario is described.

The context of this scenario is after the delivery and deployment of the Agrobot to the final users (i.e. the farmers). In this scenario, the development/maintenance team has detected that a critical update is required: the image processing algorithms do not behave correctly in some situations (e.g. when the light levels are low). These algorithms are located in the *ImageProcCard* and *ImageProcSoftware* component types. The update must be propagated and installed transparently, without disrupting the operations that the robot is carrying on (i.e. at runtime), and without any direct user operation (i.e. remotely, in an automated way). Since the *ImageProcCard* component is a hardware element that cannot be remotely changed, the maintenance team has decided to update dynamically only the *ImageProcSoftware* type, which will be automatically instantiated in these situations where the old image processing algorithms are not appropriate (e.g. by means of dynamic reconfiguration processes)

To update the image processing algorithm, an updating artefact should be sent to each Agrobot (through the *CommunicationSystem*), and perform the following actions:

1. Autenticate itself as a valid element of the system and with permissions for changing the system, so that dynamic updating services are made available. Otherwise, any element could change (and corrupt) the entire system.
2. Locate the target type that is going to be updated, and in which contexts it is being used (i.e. if it is part of other types). In this case,

the image processing algorithms are defined in the architectural type *ImageProcCard*, which is imported by the *VisionSystem* composite type.

3. Perform the updating of the type and its different instances (i.e. *RightCamera* and *LeftCamera*). In some cases, an updating may also require updating the interacting elements so that they use correctly the new behaviour. For instance, if the image data type is changed (i.e. the format used to encode images), this will also impact all the elements that manipulate these images. For simplicity reasons, we suppose that only a type needs to be updated. The updating of several types at runtime can be performed as a sequence of individual evolutions.
4. Analyse that the results are the expected, or undo the changes and return to the initial state.

It is important to note that the dynamic updating process will be able to be performed in parallel to other tasks, such as weather monitoring, energy management or surveillance. For instance, in the latter case the surveillance can be performed by one camera while the processing algorithms of the other are being updated, and so on. This is possible because in Dyanmic PRISMA, dynamic evolution is asynchronously performed (see section 7.3.4, page 266).

5.6 Conclusions

This chapter has introduced Agrobot, an autonomous agricultural robot for plague control. Agrobot is the case study that is used through the thesis to illustrate the main concepts of the approach.

The use of a case study from the domain of autonomous robotics has been motivated by the fact that the development of autonomous robots requires to deal with dependability and adaptability. Autonomous robots are complex systems that should be capable to deal with ever changing situations, most of them unforeseen (e.g. unexpected failures, new conditions, etc.), and without the direct assistance of a human once it is operating. This chapter has presented the advantages that dynamic evolution and reconfiguration can provide to the development of autonomous robots: the possibility to autonomously change their structure and behaviour at runtime, while operating and without the direct human assistance. These advantages are illustrated by means of two possible usage scenarios: dynamic updating and fault-tolerance.

In addition, this chapter has described the software architecture of the Agrobot, in terms of the PRISMA ADL, and the details of one of its subsystems, the vision system. This has provided an overview about how complex systems can be defined through PRISMA: the structure is specified by means of simple and composite architectural types, and the behaviour by means of aspects, weavings and ports.

Next, the following chapters describe how the proposed framework, called Dynamic PRISMA, supports dynamic reconfiguration and evolution and how the Agrobot uses this mechanisms.

CHAPTER VI

AUTONOMIC RECONFIGURATION

6.1 Introduction

The increasing complexity of software systems is encouraging the development of self-managed software architectures, that is, systems capable of reconfiguring their structure at runtime to fulfil a set of goals (Kramer & Magee, 2007). Several approaches have dealt with different aspects of their development, but some issues remain open, such as the scalability and maintainability of the self-management subsystem (see section 4.2.3, page 110).

To deal with these issues, this chapter presents a reconfiguration management model which is characterised by: (i) providing each composite component with self-management properties to autonomously reconfigure its internal composition, and (ii) isolating and encapsulating self-management properties into different aspects. In this sense, since self-management properties are distributed among the different components of a system and explicitly separated from other concerns, this benefits the scalability and maintenance of the self-management behaviour of the system. This approach has been called **Aspect-Oriented Autonomic Reconfiguration**, since local autonomy for dynamic reconfiguration is provided for each composite component, and separation of concerns is provided by means of Aspect-Oriented Software Development techniques (Kiczales et al., 1997). Dynamic reconfiguration is addressed in a platform-independent way, by identifying the high-level features a reconfigurable technology should provide.

This chapter is organized as follows: section 6.2 presents an overview of the main characteristics of the approach. Next section 6.3 describes the reconfiguration management model, based on an aspect-oriented control loop for self-reconfiguration. Then, section 6.4 presents in detail each one of the aspects that implement this control loop. Section 6.5 presents how these aspects are integrated in a composite component. Next, section 6.6 presents an example of the execution of a reconfiguration plan and how it is supported by this approach. Finally section 0 presents the conclusions and further works.

6.2 Characteristics of the approach

The dynamic reconfiguration approach that is presented in this thesis has the following characteristics:

- Bridges the existing gap among the specification of dynamic reconfiguration and the mechanisms that support this reconfiguration;
- Provides autonomous reconfiguration management to each composite component;
- Provides composite instances with reconfiguration plasticity to tolerate changes without breaking the design decisions defined in their types;
- Supports both proactive and reactive reconfigurations, and
- Separates the reconfiguration concerns from other concerns through aspects.

Next, these characteristics and the reasons that motivated their development are presented.

A reconfiguration model bridging the gap among specifications and mechanisms

Several works have addressed the support for dynamic change from different levels of abstraction. On the one hand, a lot of works have been focused on the technical feasibility of dynamic updating (McKinley et al., 2004) (Ritzau & Andersson, 2000) (Segal & Frieder, 1993). For this reason, these works are generally tied to a specific technology: their reconfiguration specifications are specified at a low abstraction level. On the other hand, other works have been focused on the specification of dynamic reconfigurations at a high abstraction level (i.e. by means of ADLs): (Bradbury et al., 2004), (Canal et al., 1999), (Cuesta et al., 2004), (Endler & Wei, 1992). However, most of these works have not addressed how to support the execution of such high level reconfigurations.

Since the dynamic reconfiguration of software systems is highly related with the management of running software artefacts, it should be considered not only the specification of how a system should be reconfigured, but also the mechanisms that support this reconfiguration. There is a gap among the approaches that cope with high-level reconfiguration specifications and the approaches that cope with the low-level mechanisms supporting reconfiguration. One of the major contributions of this work is the definition of a reconfiguration model that bridges this gap.

Autonomous reconfiguration management

A characteristic of this approach is that each composite instance is provided with autonomous self-reconfiguration capabilities. This allows each composite instance to reconfigure autonomously its internal composition, in response to either internal or external events, without the direct supervision of a centralized reconfiguration manager.

As it has been described in section 4.2.3.1 (see page 110), the use of a centralized reconfiguration manager to supervise and change the whole system (e.g. the self-adaptive systems of (Oreizy et al., 1999), (Dashofy et al., 2002), (Garlan et al., 2004)) has the disadvantage of decreasing the scalability of the self-management subsystem. The bigger the system is, the larger and complex the centralized reconfiguration manager is, and so its maintenance. In addition, this manager also becomes a *single-point-of-failure*: if such centralized reconfiguration manager fails, the whole system also loses its ability to reconfigure.

By providing self-reconfiguration properties to each composite instance, reconfiguration management can be distributed hierarchically among the different subsystems (i.e. composite components), thus alleviating the system from the need of a global reconfiguration manager[28]. In this way, since reconfiguration management is decentralized, the scalability of the system increases.

Type-constrained reconfigurations

Another characteristic of this approach is that it uses type-level constraints to define the *reconfiguration plasticity* of composite instances, i.e. the degree on which composite instances can change their architecture. This avoids that self-reconfigurable composite instances, due to several reconfigurations, could break the original design decisions and thus their integration with other

[28] Only in these cases where a centralized control of reconfigurations is needed, a top-level reconfiguration manager may be used, which supervises the execution of lower-level reconfiguration managers. For further information, see (Costa-Soria et al., 2011)

elements of the system. This has been performed by means of type-constrained reconfigurations.

PRISMA composite types define type-level constraints (called patterns) in terms of: what kind of architectural types can be instantiated, how many of them can be instantiated, and how they can be connected (see section 2.4.4, page 53). These type-level constraints have been used to define the reconfiguration plasticity of composite instances: a composite instance can reconfigure its internal architecture, but only while keeping the conformance to the pattern defined in its composite type.

Proactive and reactive reconfiguration support

Another characteristic that has been taken into account in this approach is the support for both kinds of activeness (Buckley et al., 2005): proactive (i.e. programmed) reconfigurations and reactive (i.e. ad-hoc) reconfigurations. Proactive reconfigurations are those architectural changes that are driven autonomously by the system when some specific conditions or events (previously defined at design-time) apply (for more details, see section 3.3.4.1, page 71). Reactive reconfigurations are those architectural changes that are driven at runtime by an external agent. Both reactive and proactive reconfigurations are complementary: they must be supported to allow a software system to reconfigure itself autonomously (e.g. by using programmed reconfigurations) and to introduce unforeseen changes or updates (i.e. ad-hoc reconfigurations). This provides the architect with a high level of flexibility for defining reconfigurable systems.

This proposal integrates both kinds of activeness by separating reconfiguration requests from reconfiguration mechanisms and making explicit their interactions. Proactive reconfigurations are encoded as part of the system, by means of Event-Condition-Action rules. By the contrary, reactive reconfigurations are externally provided, by means of reconfiguration actions. Both proactive and reactive reconfigurations use the underlying reconfiguration mechanisms, but they differ in that reactive reconfigurations can be filtered through reconfiguration ports. The architect may limit the set of reconfiguration actions that are provided by these ports and that can be accessed from outside. In this way, the architect decides which kinds of reconfigurations are provided, by appropriately connecting the reconfiguration mechanisms to other components of the system.

Separation of reconfiguration concerns through Aspects

Finally, an important characteristic of this approach is the explicit separation of reconfiguration concerns from other concerns. *Separation of concerns*

(Dijkstra, 1974) is an important principle that promotes the subdivision of a problem (e.g. the design of a software system) into independent parts (e.g. modules, components, etc.). In the context of software evolution, separation of concerns is a recommended practice to separate the parts of the software that may be subject to different rates of change (Mens & Wermelinger, 2002). This helps to avoid the entanglement of the different concerns of a software system, and improve its design and maintainability. Examples of the concerns that can be identified in a software system are: functionality, coordination, persistence, distribution, security, presentation, or even evolution and reconfiguration.

Aspect-Oriented Software Development (Kiczales et al., 1997) emerged as another approach for realizing the separation of concerns, but showing a key difference: it focuses on those concerns that crosscut a software system, facilitating that each concern can be separately specified into separate entities called **aspects**. This separation avoids the tangled concerns of software, allowing the reuse of the same aspect in different entities of the software system as well as its maintenance.

Several proposals have addressed the integration of aspects in software architectures (Cuesta et al., 2005). However, very few of them (David & Ledoux, 2006), (Batista et al., 2008) have considered the importance of encapsulating the reconfiguration concern into aspects. We consider that *the separation among reconfiguration concerns and the other concerns of a system is a first step to build adaptive systems easier to maintain*. Thus, the reconfiguration code will be able to change the functional code without being affected. Therefore, this approach takes advantage of AOSD techniques to improve the reconfiguration management.

6.3 Reconfiguration management model

An aspect-oriented control loop for self-reconfiguration

The approach that is proposed in this thesis provides composite components with proactive capabilities for reconfiguring their composition. This means that a composite instance is able to autonomously perform changes to itself when some specific conditions or events apply. This requires the availability of some *control system* (see section 3.5.1, page 84) to supervise the execution of the composite instance and perform the reconfiguration actions when needed. Since this control system implements the reconfiguration concern of a composite instance, it has been encapsulated into aspects.

This section presents the control system that has been defined for the autonomous management of reconfigurations, and the reasons that have motivated its encapsulation into aspects.

6.3.1 A control loop for self-reconfiguration

As it has been stated by other authors (Oreizy et al., 1999) (Bradbury et al., 2004) (Garlan et al., 2004) (Greenwood & Blair, 2006) (Kramer & Magee, 2007), self-managed architectures generally follow a closed control loop that periodically supervises the architecture, plans if any (corrective) change needs to be performed, and effects them.

Similar control loops have been proposed to develop autonomous systems (e.g. robots), being the most extended the autonomic control loop (Kephart & Chess, 2003), which is usually referred to as the MAPE-K loop (Monitor, Analyse, Plan, Execute, Knowledge). This loop performs control operations on a managed resource to achieve a set of predefined high-level goals, which are part of the knowledge of an autonomic (i.e. self-controlled) element. The autonomic control loop has the advantage that clearly isolates the main concerns commonly present in every process of (self-)change. Other architecture-based proposals for self-management generally merge analysis and planning, or planning and execution, or do not explicitly model the knowledge required to perform the changes.

The approach that is proposed in this thesis uses the autonomic control loop as a reference model to define how a system reconfigures itself, bridging the gap among high-level specifications (i.e. ADLs) and technology-specific (dynamic updating) mechanisms. The original MAPE-K loop has been adapted for this purpose: the managed resource is the architecture of a system, and the control operations performed on this resource are mainly introspection operations (for monitoring the architecture), and reconfiguration operations (for changing the architecture).

Another adaptation that has been done to the original MAPE-K loop is its implementation through *aspects* instead of *modules* (the reasons that have motivated this decision are described in section 6.3.2). Each one of the different controlling components (i.e. Monitor, Analyse, Plan, etc.) has been encapsulated in a different aspect. These aspects, and the concern they encapsulate, are the following (see Figure 6.1):

1. *Monitoring*, the crosscutting concern that captures the events that take place in the architecture of a system (i.e. the managed resource);

2. *Reconfiguration Analysis*, the concern that analyzes the different events to detect if a reconfiguration must be done, and which defines the set of reconfigurations which must be performed on the architecture;
3. *Reconfiguration Coordination*, the concern that plans/coordinates how the reconfigurations must be applied safely on the architecture without interrupting current transactions, and
4. *Reconfiguration Effector*, the concern that applies atomic reconfiguration operations on the running system.

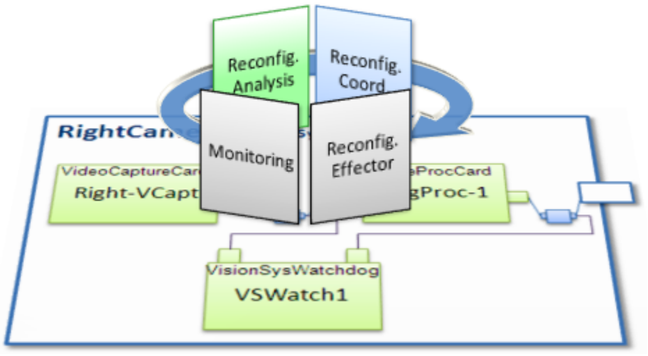


Figure 6.1. Aspects for autonomic reconfiguration

Note that, in our proposal, the Knowledge element of the original MAPE-K loop is not required. In other architecture-based approaches, the Knowledge element has been generally used to maintain a runtime model of the architecture, which is analysed to decide whether reconfigurations are needed or not (Huebscher & McCann, 2008). However, in our proposal there is no need for an explicit management of a runtime model of the system. All the information about the architecture of the running system (e.g. the status of its architectural elements, types and configurations) is provided by means of reflection techniques, through the Monitoring aspect. This has the advantage that the architectural information is always updated, reflecting the real status of the system.

Other works have used the Knowledge element for the storage of useful information for triggering or generating reconfiguration plans (e.g. use of cumulative state information about the architecture, use of utility functions, etc.). The Knowledge element keeps the collected information, which is analyzed by the Analysis element. However, in our proposal, the information for triggering or generating reconfiguration plans is included as part of the Reconfiguration Analysis aspect, instead of in a Knowledge element. This is because this information belongs to the same concern that the Reconfiguration Analysis aspect defines: the definition of *proactive* *reconfiguration behaviour*. Thus, the Reconfiguration Analysis aspect encapsulates all the required information and algorithms for triggering and generating high-level reconfiguration plans.

6.3.2 Aspects versus modules

The approach that has been proposed in this thesis adapts the original MAPE-K control loop through aspects instead of modules. The reason to use aspects and not modules for encapsulating dynamic reconfiguration behaviour is because of the advantages that AOSD provides (Kiczales et al., 1997): better reuse and maintenance of the different concerns.

Although modules can be used to separate concerns, the invocations among different modules (e.g. procedure calls) are defined explicitly inside each module, thus making each module dependent of the other. However, in the aspect-oriented model defined by the PRISMA approach, aspects are, by definition, independent of each other. In PRISMA, we cannot talk about *invocations* among aspects, but *synchronizations* among aspects. An aspect defines provided and required services, and each service is treated as a *hook* which can be intercepted. These interceptions are performed by weavings, which are defined outside the aspects and define how two aspects are bound together (i.e. synchronized). In addition, an advantage of using weavings to hook services among aspects is that they do not need to have the same name and signature, since functions can be used to adapt incompatible signatures (Guillén-Martín, 2007; pp. 43-60). Thus, aspects are completely independent of each other: modifying an aspect will only impact the weavings that are specifically related to this aspect, but not other aspects.

For instance, Figure 6.2 shows some of the weavings that have been defined in the VisionSystem architectural type for synchronizing the different reconfiguration aspects. The first weaving intercepts the execution of the service *beforeEvent* (provided by the Reconfiguration Analysis aspect), and replaces it with the execution of the service *beforeServiceRequest* (provided by the Monitoring aspect). In this way, the Reconfiguration Analysis aspect and the Monitoring aspects are bound together, without one aspect explicitly declaring a reference to the other. The weaving captures the callings to the *beforeEvent* service and replaces them by the *beforeServiceRequest* service.

In case any of the services that take part in a weaving has its signature changed (e.g. due to a maintenance operation), then the weavings where it participates are invalidated. This results in that weaved services are unbound (i.e. the matching among services do not occur). Therefore, the modification of an aspect does not necessarily impact the other aspects. The analysis of this impact is outside the scope of this work; however other authors have addressed conveniently this problem, such as (Perez-Toledano et al., 2007).

```
Weavings
   ...
   Monitoring.beforeServiceRequest(*, eventName, eventParams)
      insteadOf
   VisionSystemReconfigurationAnalysis.beforeEvent(eventName,eventParams);

   Monitoring.getArchElementInstances("VideoCaptureCard", list-IDs)
      insteadOf
   VisionSystemReconfigurationAnalysis.
         getInstances-videoCaptureCard(list-IDs);

   VisionSystemReconfigurationServices.create-ImageProcSoftware(params,
              newID)
      insteadOf
   VisionSystemReconfigurationAnalysis.create-ImageProcSoftware(params,
              newID);

   // ... more weavings

End_Weavings;
```

Figure 6.2. Example of weavings among aspects

6.4 Description of the autonomic reconfiguration aspects

In this section, the different aspects of the approach that have been defined to support autonomic reconfiguration are described in detail. This approach defines four concerns that are encapsulated into aspects. These aspects have been defined to separate reconfiguration specifications from reconfiguration mechanisms, and thus increase their maintainability. This avoids that changes on reconfiguration mechanisms (which are technology-dependent) may impact reconfiguration specifications (which are technology-independent), and viceversa.

Each aspect has a different role in our approach (see Figure 6.1 in page 165). On the one hand, the *Reconfiguration Analysis* aspect is domain-specific: it is defined by the architect and contains the reconfiguration policies that are specific for the composite component that it is weaved to. These policies are defined using high-level terms, i.e. using PRISMA concepts, thus avoiding the use of low-level details. On the other hand, the *Monitoring* and *Reconfiguration Effector* aspects (depicted in grey in Figure 6.1) are technology-dependent: they implement the mechanisms that provide support for supervising and changing the system architecture. They model the low-level services that are provided by the infrastructure and allow us to combine them to perform high-level reconfiguration operations. This combination is performed by the *Reconfiguration Coordination* aspect: it encapsulates the mappings from high-level PRISMA concepts to low-level technological services. Thus, the code that has different rates of change (Mens & Wermelinger, 2002) is explicitly separated: dynamic updating mechanisms (i.e. *Monitoring* and *Effector* aspects), reconfiguration specifications (i.e. *Analysis* aspect), and the mappings among them (i.e. *Coordination* aspect).

6.4.1 The Monitoring Aspect

Proactive reconfigurations are generally triggered when a specific event, error or state is detected in the architecture of a system. To detect these events, errors or states, a mechanism that continually monitors the architecture at runtime is needed. This aspect *monitorizes* the architecture of the composite instance where it has been imported to. The PRISMA specification of the Monitoring aspect is shown in Figure 6.3.

The Monitoring aspect has access to the internal structure of a composite instance and provides information about it, which can be used for different purposes (e.g. introspect the architecture, initiate a reconfiguration process, detect events, etc). The internal structure of a composite instance (i.e. its architecture) is composed of: architectural elements (i.e. Components, Connectors or Systems), links, and ports. *Components and Systems* process services, and *Connectors* coordinate services; *links* forward services among architectural elements (i.e. Attachments) or ports (i.e. Bindings); and *ports* forward service requests to external architectural elements. This structural information is provided through a set of attributes (see Figure 6.3): (i) *systemID*, a reference to the composite instance being monitored; (ii) *architecturalElements*, *systemPorts*, *attachments* and *bindings*, the references to the structural parts of a composite instance; and (iii) *monitoredServices*, the services that are being monitored or intercepted. These attributes always reflect the current structure and state of the composite instance (they are updated by the underlying middleware). However, these attributes cannot be directly accessed: the information stored on them can be only obtained by using the different services provided by the Monitoring aspect.

```
ArchitectureMonitoring Aspect
   using I_SystemInstanceIntrospectionServices

// ***************************************************************
// Platform-dependent aspect for monitoring and introspecting
// System instances at runtime
// Here are described the public services that are provided.
// Internal behaviour is provided by low-level implementations.
// CANNOT BE CHANGED BY THE USER
// ****************************************************************

   Attributes
      Constant
         systemID: string; // Reference to the System instance
      Variable
         architecturalElements: list;
         systemPorts: list;
         attachments: list;
         bindings: list;
         monitoredServices: list;

   Services
         // ******** Introspection services ********
1  in getConfigurationSpecification(output PRISMAConfSpec: string);
2  in typeOf(instanceID: string, output typeName: string);

3  in getArchElementInstances(typeName: string,
         output instances: list);
4  in getAttachedArchElems(archElemID: string, attachType: string,
         output attachedArchElemIDs: list);
5  in getConnectionsOfArchElem(archElemID: string,
         output connectionList: list);
6  in getConnectionsByType(connectionType: string,
         output connectionList: list);
7  in isAttachment(connID: string, boolean isAtt);
8  in isBinding(connID: string, boolean isBind);

9  in getArchElementProperties(archElemID: string,
         output properties: list, output portsList: list);
10 in getPortProperties(archElemID: string, portName: string,
         output isProvided: boolean, output isRequired: boolean,
         output interface: string, output connectionList: list);
11 in getArchElementInitializationValues(archElemID: string,
         output initValues: list);
12 in getAttachmentProperties(connectionID: string,
         output instance1: string, output instance2: string);
13 in getBindingProperties(connectionID: string,
         output sysPort: string, output archElemID: string);

   // ******** Runtime Status Information ********
14 in getStatus(elementID: string, output status: string);
15 in getElementsOfStatus(status: string, output elemIDList: list);

   // ******** Event Interception Services ********
16 in beforeServiceRequest(elemID: string, serviceName: string,
         output params: list);
17 in afterServiceRequest(elemID: string, serviceName: string,
         output params: list);
18 in insteadOfServiceRequest(elemID: string, serviceName: string,
         condition: string, replacingService: string,
         output params: list);
19 in monitoredServices(output serviceList: list);
End Monitoring Aspect;
```

Figure 6.3. Services of the Monitoring aspect

The Monitoring aspect provides a set of services that collect different kinds of information:

- The current *Configuration* of the architecture,
- The *Runtime Status* of the different elements of the architecture,
- *Events* that take place in the architecture.

These services are described in detail in the following subsections.

6.4.1.1 Introspection Services

Since the architecture of a dynamic reconfigurable system can change substantially over time, information about the configuration at any given moment is essential. Introspection services provide information about the actual configuration of the managed architecture.

At runtime, an architecture configuration is described by a set of references (or runtime IDs) to the architectural element instances (i.e. component and connector instances), and a set of references (connection IDs) to the established links (i.e. attachments and bindings) among the architectural element instances. There are two ways for retrieving the configuration of a composite instance: through the generation of a PRISMA model describing the actual architecture, or through the use of introspection services to obtain actual information about each element of the architecture. Each one is described next.

A service to obtain a snapshot of the architecture

The Monitoring aspect provides a service that returns a model (or snapshot) describing the actual architecture of the composite instance:

- `GetConfigurationSpecification`. This service returns a PRISMA specification that contains a snapshot of the different architectural elements that are instantiated in the composite instance and how they are connected to each other.

For instance, given the PRISMA configuration specification of *RightCamera* (see Figure 5.7), this service would return the specification depicted in Figure 6.4. Note the different elements contained in the specification: (i) the kind of PRISMA element (i.e. component, connector, attachment or binding); (ii) the identifier (i.e. attribute "id"); (iii) the type; and (iv) initialization values (e.g. "initParameters" for Components and Connectors, attributes "source" and "target" for attachments, attribute "target" for bindings, etc.).

```
<Configuration id="RightCamera" type="VisionSystem">

   <Component id="Right-VCapt" type="VideoCaptureCard">
      <InitParameter>30</InitParameter>
   </Component>
   <Component id="ImgProc-1" type="ImageProcCard">
      <InitParameter>Right</InitParameter>
   </Component>
   <Component id="R-ImgWatchDog1" type="VisionWatchdog">
      <InitParameter>90</InitParameter>
   </Component>
   <Component id="R-Evolver" type="VisionSystemEvolver">
      <InitParameter>RightCamera</InitParameter>
   </Component>

   <Connector id="VCC-Conn1" type="VCC-Conn" />
   <Connector id="IPC-Conn1" type="IPC-Conn" />

   <Attachment id="att1" type="Att_VCC_VCCConn"
      source="Right-VCapt" target="VCC-Conn1" />
   <Attachment id="att2" type="Att_VCCConn_IPC"
      source="ImgProc-1" target="VCC-Conn1" />
   <Attachment id="att3" type="Att_IPC_IPCConn"
      source=" ImgProc-1" target="IPC-Conn1" />
   <Attachment id="att4" type="Att_VCCConn_ImgMon"
      source="VCC-Conn1" target="R-ImgWatchDog1" />
   <Attachment id="att5" type="Att_IPCConn_ImgMon"
      source="IPC-Conn1" target="R-ImgWatchDog1" />
   <Attachment id="att6" type="Att_ImgMon_Evolver"
      source="R-ImgWatchDog1" target="R-Evolver" />

   <Binding id="bin1" type="Bin_IPCConn" target="ImgProc-1" />
   <Binding id="bin2" type="Bin_Evolver" target="R-Evolver" />

</Configuration>
```

Figure 6.4. Example of PRISMA XML Configuration Model

The advantage of retrieving a complete snapshot of the architecture is that all the structural information of a composite instance is provided in a single message (i.e. the execution of a service), and that it can be computationally processed (e.g. if this model is provided in XML). This model can be processed for different purposes:

- To generate a visual representation of a composite instance, like the shown in Figure 5.8, that may be used to understand/simulate the current state of a composite.
- To analyse the architecture and decide if reconfigurations are needed (i.e. reactive behaviours/ad-hoc reconfigurations)

The disadvantage of retrieving such a complete snapshot is that additional modules for reading and interpreting the snapshot are needed, in addition to the functionality that it is going to use this snapshot. In case simple queries are needed, such as obtaining the attached elements of an architectural element, the use of introspection services are more appropriate.

Services to query the actual configuration

To retrieve specific, concrete structural information from a composite instance without the need of analysing the entire model, a set of additional introspection services are provided. These services allow us to get the instances of a particular type, the connections to a given instance, their properties, etc. These services are described below:

- `TypeOf`: This service returns the type name of a given instance ID. This not only allows us to identify the type of an instance, but also to get access to: (i) the constraints and properties defined by its type (such as the minimum and maximum cardinality) or (ii) the *reify* service, which allows us to evolve its type dynamically, as it will be described later (see Chapter 7).
- `GetArchElementInstances`: This service returns a list, *instances*, with the IDs of the instances of a concrete type, *typeName*. In addition, if *typeName* is a null string "", or the wildcard symbol "*", then the service returns the IDs of all the instances created inside the composite instance, independently of their types. This service is useful for retrieving the list of instances that have been created in a composite instance, or only the instances of a certain type.
- `GetAttachedArchElems`. Given the ID of an architectural element instance, *archElemID*, and the type of an attachment, *attachType*, this service returns a list, *attachedArchElemIDs*, with the IDs of all the instances that are attached to *archElemID* following the interaction pattern *attachType*. In addition, if *attachType* is a null string "", or the wild card symbol "*", then the service returns *all* the instances that are attached to *archElemID*, independently of their attachment type. This service is useful for retrieving the architectural elements that are attached to another, or only those that follow a concrete interaction pattern.
- `GetConnectionsOfArchElem`. Given the ID of an architectural element instance, *archElemID*, this service returns a list with all the connections (i.e. attachments and bindings) that are linked with *archElemID*. In addition, if *archElemID* is a null string "", or the wild card symbol "*", then the service returns *all* the connections that have been created in the composite instance. Note the difference with respect to the previous service: `getAttachedArchElems` returns a list with the IDs of attached architectural elements, and

getConnectionsOfArchElem returns a list with the IDs of connections. This service is useful for obtaining the set of connections/links created in the composite instance, allowing its further manipulation (e.g. recreating them after an architectural instance replacement)

- GetConnectionsByType. This service returns all the connection IDs of a given connection type (i.e. an attachment type or a binding type).
- IsAttachment and IsBinding. These services allow us to distinguish if a given connection ID is an attachment or a binding (which have different properties).

The services described above allow us to query the configuration of a composite instance and retrieve the IDs of its elements (either architectural elements or connections).

An additional set of services allow us to query the properties of these elements:

- GetArchElementProperties. Given the ID of an architectural element instance, *archElemID*, this service returns a list with its properties, *properties*, and a list with its ports, *portsList*. In addition, if *archElemID* is a null string "", or the symbol "self", then the service returns the externally-visible properties and ports of the composite instance.
- GetPortProperties. Given the ID of an architectural element instance, *archElemID*, and the name of one of its ports, *portName*, this service returns the properties of such port: (i) if it is a provided port (i.e. provides services to other elements, it has a server behaviour), (ii) if it is a required port (i.e. requires services from other elements, it has a client behaviour), (iii) the interface of services provided/required through the port, and (iv) the set of current connections that are linked to this port. Note that if a port has both a client and server behaviour, then both properties *isRequired* and *isProvided* will be true.
- GetArchElementInitializationValues. Given the ID of an architectural element instance, *archElemID*, this service returns the values that have been provided to its creation. This information is provided in the PRISMA ADL Configuration specification to describe how architectural elements should be created. In certain situations, this information may be useful to instantiate architectural elements with the same initial properties (e.g. in replacement operations), or to identify some required configuration values.

- `GetAttachmentProperties`. Given the ID of an attachment, *connectionID*, this service returns the IDs of the architectural element instances that are linked by means of the attachment *connectionID*. The name of the ports that an attachment connects is obtained from the attachment type.
- `GetBindingProperties`. Given the ID of a binding, *connectionID*, this service returns the name of the composite instance port and the ID of the architectural element instance that are linked by means of the binding *connectionID*.

In this way, a System instance (i.e. a composite instance) can be aware of its configuration and use this knowledge to decide if a reconfiguration is needed. Furthermore, this information also allows us to verify whether or not a set of reconfiguration actions has been successfully executed (i.e. the target configuration has been achieved).

One of the advantages of using the introspection services described above is that they provide the most updated information about the actual configuration of a composite instance. The disadvantage of working with runtime models, such as the model returned by the service *GetConfigurationSpecification*, is that it may become quickly out-of-date if the composite instance changes during the processing of the model (which may take time). By using the introspection services, we always have the updated information about the actual elements of the architecture. And, in such a case that the properties of a removed element are queried, the execution of an introspection service would fail, thus alerting the user. However, the disadvantage of using the introspection services is that they are not appropriate for exploring the entire configuration model, such as when interested on wide analysis or visual representation. In such cases, the service *GetConfigurationSpecification* should be used instead of.

6.4.1.2 Runtime Status Information

The architecture of a composite instance can only be reconfigured when the elements that are going to undergo changes, and the connections among them, are safely stopped. For this reason, the Monitoring Aspect also provides information about the runtime status of its elements. This information is mainly used by the reconfiguration mechanisms (i.e. the Reconfiguration Coordination aspect) to decide when a certain element of the architecture is ready to be changed.

This information is provided by the following services:

- `GetStatus`. Given the ID of one of the elements of a composite instance, *elementID*, this service returns its runtime status, *status*. *ElementID* can be an architectural element instance, a connection, or the name of a port of the composite.
- `GetElementsOfStatus`. This service returns all the elements of the architecture that have a certain status, which is provided in the parameter *status*. For instance, this service is useful to obtain the set of elements that are stopped.

The status of an element at a certain time is one of the following:

- *Idle*. The element is not executing (in case of an architectural element instance) or forwarding (in case of a connection or port) any service.
- *Active*. A service is being processed (in case of an architectural element) or forwarded (in case of a connection or port).
- *Blocked*. The element is in a consistent state and ready to apply runtime changes. Running transactions have been finished safely. New service requests are accepted, but they are queued until the element status is changed to the Active status.
- *Blocking*. This is a transitional status from Active status (executing a service) to Blocked status (not executing services and ready for reconfiguration). The element waits until the Tranquillity (Vandewoude et al., 2007) or Quiescence (Kramer & Magee, 1990) criterion is achieved (see section 3.4.1, page 74 for more details). These criteria guarantee that the element has reached a consistent state (tranquillity causes less disruption to the architecture than quiescence).
- *Unknown*. The element is not responding because an error has occurred. This information can be used to provide fault-tolerance mechanisms.

6.4.1.3 Event Interception Services

The Monitoring aspect has access to the set of events that take place in the architecture of a composite instance, and provides services to intercept them and act before, instead of, or after the event.

An event is an action that takes place inside a given context that others may be interested in knowing about it. In this case, the context is the (internal) architecture of a composite instance. As described previously, this architecture is composed of architectural elements that request services from each other. These architectural elements are black-boxes, and for this reason, we cannot intercept actions or events that occur inside them. Thus, the events that take place inside a composite instance, and which can be intercepted by the Monitoring aspect, are those that take place at the level of interactions: that is, *Service Requests*. In this way, the behaviour of internal architectural elements remain unaffected by interception mechanisms: they are unaware of these mechanisms. The Monitoring aspect intercepts service requests when they are "transferred" through the connections among architectural elements (i.e. attachments), and those coming from/to the external ports of the composite instance (i.e. bindings).

The Monitoring aspect provides three services to intercept events: *beforeServiceRequest*, *afterServiceRequest* and *insteadServiceRequest.* These services are blocking, that is, the caller is blocked until the required event or service request is intercepted in the architecture. Then, when the desired event or service request is intercepted, the subscriber is unblocked and can perform the required actions in response to such notification. As will be described in section 6.4.2.2, this is used to implement reconfiguration triggers, i.e. mechanisms that are only activated when a specific event (or service request) is detected in the architecture being managed.

The services *beforeServiceRequest* and *afterServiceRequest* intercept the transfer of a service request or event among a client instance, i.e. which invokes the service, and a server instance, i.e. which provides and serves the service (see Figure 6.5, left). These interceptions take place at different times: <u>before</u> the service request is delivered to the server instance (*beforeServiceRequest*), or <u>after</u> the service request is processed (*afterServiceRequest*). However, these services do not interrupt the normal execution flow among the client instance and the server instance: the notification about the occurrence of a service request is sent seamlessly to the subscriber, without affecting the delivery of the service request.

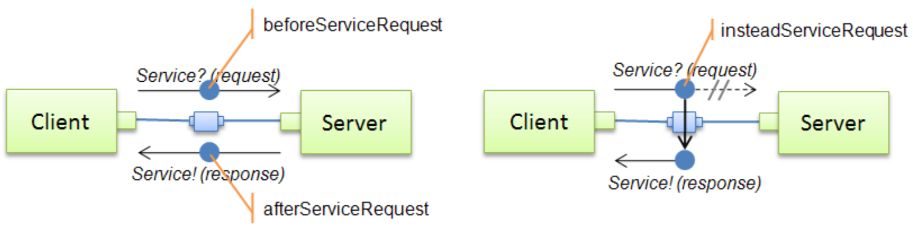


Figure 6.5. Event interception behaviour

The service *insteadServiceRequest* performs a service interception: it replaces the execution of a service request by another (see Figure 6.5, right). The target service request is intercepted when it passes through the connector and a different response is sent back to the client instance. This interception is only performed if a user-defined condition holds on the parameters of the service request. This is useful to trigger a reconfiguration when a service is requested with a set of specific parameters (e.g. invalid parameters). In order to minimize the disruption on the architecture, the result values expected by the client instance must be provided by the service *insteadServiceRequest*. Otherwise, if the service request were simply cancelled, the client instance would be indefinitely expecting results, and its execution disrupted.

The description of the event interception services, and the parameters expected, is provided below (see their signature in Figure 6.3):

- `BeforeServiceRequest`. This service sends a notification to the subscriber just <u>before</u> a service request is delivered to the service provider. The parameter *serviceName* defines the service request to intercept, whereas *elemID* defines which element sent the request (a port of the composite instance, an architectural instance or a connection). If *elemID* has the value "*", then no selection of the service requestor is performed (i.e. a notification is sent for any occurrence of *serviceName* in the architecture of a composite instance). Finally, the output parameter *params* returns the set of parameters of the service request that has been intercepted.
- `AfterServiceRequest.` This service sends a notification to the subscriber just <u>after</u> a service request is processed by the service provider. The meaning of its parameters is the same as *BeforeServiceRequest*, so they are not described again.
- `InsteadOfServiceRequest`. This service intercepts the delivery of a service request or event, notifies the subscriber about this fact, and executes a different service instead. This interception is only performed if a specific condition is satisfied by the parameters of the service request. The parameter *serviceName* defines the service request to intercept, and *elemID* the element which sent the request (i.e. a port, an architectural instance or a connection). The parameter *condition* provides a condition to evaluate on the set of parameters of the service request. The parameter *replacingService* is the name of the service that will be executed instead of the original service, and which must be provided by the subscriber. Finally, the output

parameter *params* returns the set of parameters of the service request that has been intercepted.

- `MonitoredServices`. This is an auxiliary service that returns the complete list of services that are monitored for interception. This service is useful for implementing dependency analysis: to analyze the elements of the architecture that are being supervised and that, if they are changed, may impact the reconfiguration policies.

For instance, to be subscribed to (i.e. intercept) the event *faultyOutput* when it is triggered by the *VisionSysWatchdog* component (i.e. before the event is processed), the following code must be executed:

```
beforeServiceRequest!("VisionSysWatchdog", "faultyOutput",
                              output params)
```

Thus, if the *faultyOutput* event is triggered in the architecture, then the Monitoring aspect will notify the caller about this fact, providing the parameters of the intercepted event.

Moreover, reconfiguration services (see section 6.4.3.3) can also be intercepted, as they take place in the composite instance boundary. For instance, the following code will capture the event of creating an architectural element instance immediately after being processed:

```
afterServiceRequest(null,"CreateInstance",output creationParams)
```

In this example, the parameter *elemID* is null because reconfiguration services are not requested through any element of the architecture (they are requested at the meta-level). The parameters of the service will be returned in the variable *creationParams*, and we could get the name of the type that has been instantiated, its initialization parameters, and the ID of the new instance.

The interception of reconfiguration events is a very useful feature to keep visual representations or models of a system updated as soon as the system changes. If an architectural instance is created, removed, or its connections changed, then the visual model should be notified to represent this situation. This is better than periodically obtaining the complete specification of the running system (by means of the *getConfigurationSpecification* service) and checking changes.

6.4.2 The Reconfiguration Analysis Aspect

The *Reconfiguration Analysis* aspect describes the *proactive*[29] reconfiguration behaviour of the composite component that it belongs to. This aspect is application-specific: it is defined for a specific composite component, and contains the policies or goals that drive the reconfiguration of this component. This aspect defines *when* to perform a reconfiguration, and *how* the different architectural elements must be reconfigured.

It is important to emphasize that this work is only concerned with the correct execution of dynamic changes, but not with trying to establish whether the changes proposed are correct. For this reason, the reconfiguration plans defined in this aspect must be correct and valid, which is a responsibility of the architect. The reconfiguration infrastructure is only responsible for guaranteeing that the changes requested are correctly introduced. Syntactical analysis are performed to check whether the resulting configuration is correct (i.e. all the elements are well connected), but not a behavioural analysis.

The *Reconfiguration Analysis* aspect is a new kind of aspect that has been introduced in the PRISMA metamodel: it is defined as an automatically-generated template that the user (i.e. the system architect) completes, by defining Event-Condition-Action (ECA) policies for reconfiguration. The user defines ECA policies by means of: *reconfiguration triggers* (i.e. Events and Conditions) and *configuration transactions* (i.e. Actions).

In order to illustrate how these policies are defined, we will use the fault-tolerance reconfiguration scenario defined in the *VisionSystem* type (see section 5.5.1).

6.4.2.1 Structure of the Reconfiguration Analysis Aspect

The Reconfiguration Analysis aspect has the same structure as other kinds of PRISMA aspects, with subtle behaviour differences. The generic structure of the Reconfiguration Analysis aspect is shown in Figure 6.6. An example of such kind of aspect is shown in Figure 6.7. The complete specification of this example is provided in section A.2.3.

```
ReconfigurationAnalysis Aspect <Aspect_Name>
   using <Used_Interfaces>
   is partially defined by <SystemTypeName>AnalysisServices

   Attributes
      Constant
```

[29] See section 3.3.4.2 on page 68 for a description of the kinds of activeness.

```
            <User-defined constants>
         Variable
            <User-defined attributes>
         Derived
            <User-defined derived attributes>

      Services
         begin(<additional_initialization_params>);
         end();

         <User-defined services and events>

      External Functions
         <External function headers>
      End_External Functions

      Played_Roles
         <Player_Roles: internal/external communications>

      Triggers
         <RECONFIGURATION TRIGGERS: Events and Conditions>

      Transactions
         <CONFIGURATION TRANSACTIONS: Actions to perform in the arch>

      Protocol
         <Initialization of default parameters, definition of
processes, and orchestration of configuration transactions>

End ReconfigurationAnalysis Aspect <Aspect_Name>;
```

Figure 6.6. Pattern of the Reconfiguration Analysis Aspect

An aspect of this kind has a (user-defined) unique name <*Aspect_Name*>, and a set of user-defined provided/required interfaces <*Used_Interfaces*>. The aspect is partially defined: it has a user-defined part and an automatically-generated part, which provides the hooks to weave this aspect with the other reconfiguration aspects. This automatically-generated part is named following this pattern: <*SystemTypeName*>*AnalysisServices.* This will be described in section 6.5.3.

The **Attributes** section contains user-defined constants, variables, and derived attributes (i.e. calculated on demand applying a derivation rule or function). The architect can easily define new attributes to capture certain properties from the architecture, such as performance. For instance, the derived attribute *imageProcPerformance* (see Figure 6.7, *Derived* subsection) returns the ratio of images that are processed per second in the *VisionSystem* (the higher the best). This is calculated by an external function, not described here, called *FCalculateImageProcRatio*.

The **Services** section contains at least the initialization and destruction services *begin* and *end*, respectively. The *begin* service is generally used to get initialization values. For instance, in the VisionSystem example, the begin service provides the camera position, which can be "left" or "right" (see Figure 6.7, *Services* section). In case the aspect receives or sends events to other elements, the headers of these events are defined in this section, specifying if they are incoming or outcoming events. For instance, in the VisionSystem example, this aspect receives events from internal elements (i.e. the events *faultyOutput* and *validOutput*, from the VisionWatchdog component), and sends events to external elements (i.e. the events *enabledSystem* and *disabledSystem*, to other subsystems of the Agrobot).

The **External Functions** section defines the headers of externally defined functions, usually low-level functions that are defined/provided by the target platform. For instance, the aspect of the example defines a function to suspend the execution a certain time.

The **Played_Roles** section is related to the Services section, since it defines the different roles on which the different services can be received and/or sent. For instance, the example defines two roles: `INTERNAL-EVENTS`, for the services that are received, and `EXTERNAL-EVENTS`, for the services that are requested to other elements (i.e. sent events) (see Figure 6.7, *Played_Roles* section)

```
ReconfigurationAnalysis Aspect VisionSystemReconfigurationAnalysis
   using I_WatchdogEvents, I_VisionSystemEvents
   is partially defined by VisionSystemAnalysisServices

// ******************************************************************
// User-defined aspect which contains the reconfiguration policies
// for VisionSystem instances.
// ******************************************************************

   Attributes
      Constant
         // User-defined constants here
      Variable
         cameraPos: string;
      Derived
         imageProcPerformance: integer,
            derivation: FCalculateImageProcRatio();

   Services
      // Initialization and destruction services
      begin(cameraPosition : string)
         Valuations
         [begin(cameraPosition] cameraPos = cameraPosition;
      end();

      // Interaction with the VisionWatchdog component
      in faultyOutput(failingComponent: string);
      in validOutput();

      // Notification of critical events to external elements
      out disabledSystem(instanceID: string, reason: string);
      out enabledSystem(systemID: string)

   External Functions
      Suspend(timeout : integer); // Suspends the current process
   End_External Functions


   Played_Roles
      INTERNAL-EVENTS for I_WatchdogEvents ::=
         faultyOutput?(failingComponent) + validOutput?();

      EXTERNAL-EVENTS for I_VisionSystemEvents ::=
         disabledSystem!(instanceID, reason) +
         enabledSystem!(instanceID);

   Triggers
      ...

   Transactions
      ...

   Protocol
      VISIONRECONFANALYSIS ::=  begin(cameraPosition) --> ANALYSIS;

      ANALYSIS ::= end():10
      +  (  AddImageProcessor():10 ) --> ANALYSIS
      +  (  RepairImageProcessingUnit():10 ) --> ANALYSIS
      +  (  DisableVisionSystem():20 ) --> DISABLEDSTATE;

      DISABLEDSTATE ::= Suspend(1200) -->
         incrementalStart(output success) -->
         if (success==false) then DISABLEDSTATE
         else (EXTERNAL-EVENTS_enabledSystem!(SystemID)-->
             ANALYSIS);

End ReconfigurationAnalysis Aspect
VisionSystemReconfigurationAnalysis;
```

Figure 6.7. Example of a Reconfiguration Analysis Aspect (fragment)

Finally, the **Protocol** section defines the different execution states of the Reconfiguration Analysis aspect and how the different services (and configuration transactions) are orchestrated. In addition, this section also initializes the default parameters for instantiating architectural elements. For instance, the VisionSystemReconfigurationAnalysis aspect defines three processes: `VISIONRECONFANALYSIS`, `ANALYSIS` and `DISABLEDSTATE`. The former is the initial state, which starts when the begin service is executed. Then, the default parameters of the components imageProcCard and imageProcSoftware are set. The next state, `ANALYSIS`, waits until: (i) the end service is executed (i.e. the VisionSystem is shut down); (ii) a configuration transaction is executed (then remaining in the *Analysis* state); or (iii) the *DisableVisionSystem* configuration transaction is executed (which disables all the elements of the VisionSystem), changing to the state `DISABLEDSTATE`.

Finally, in this state, the process is suspended and a restarting is tried. If a success is achieved, the *enabledSystem* event is sent to outside (i.e. the other Agrobot subsystems) and the `ANALYSIS` state is achieved. Otherwise, the `DISABLEDSTATE` process is repeated.

Next, the reconfiguration triggers and configuration transactions are described in detail.

6.4.2.2 Reconfiguration Triggers

A *Reconfiguration Trigger* is a condition which, if true, activates a configuration transaction. This condition may evaluate user-defined attributes (e.g. performance), or be true when a certain event is intercepted (e.g. an exception, a service invocation, the creation or destruction of connections, etc.). The syntax of triggers has the following structure:

```
<NameOfConfigurationTransactionToExecute> when <Condition>;
```

Where <*condition*> can be of three kinds: attribute evaluation, service invocation or event interception.

Attribute evaluation. A user-defined attribute is evaluated, and this evaluation may trigger a reconfiguration process. For instance, if the derived attribute *imageProcPerformance* (see Figure 6.8, *Derived* subsection) returns a value lower than 10 (images per second), a reconfiguration trigger is activated (see Figure 6.8, *Triggers* section), which executes a configuration transaction for creating additional image processing instances.

```
ReconfigurationAnalysis aspect VisionSystemReconfigurationAnalysis
...
   Attributes
      Derived
         imageProcPerformance: integer,
            derivation: FCalculateImageProcRatio();
...
   Triggers
      AddImageProcessor() when
         imageProcPerformance < 10;
...
```

Figure 6.8. Reconfiguration triggers: activation by attributes

Service invocation. The condition is true when a certain service is requested through a played_role of the aspect, that is, from the composite where the *ReconfigurationAnalysis* belongs to.

For instance, the reconfiguration trigger shown in Figure 6.9 is activated if the service *faultyOutput* is requested (note the service is postfixed with the symbol "?") AND its parameter *failingComponent* equals to "ImageProcCard". Recall that this service is invoked by the *VisionWatchdog* component when a failure is detected in the architecture of the *VisionSystem* type. This means that, in order to receive a notification of this service, the aspect must be a service provider (see the declaration of the *faultyOutput* service in the Services section, with the "in" modifier, Figure 6.9) *and* be accessible to other elements (see Figure 6.9, PlayedRoles section). This also requires that the service requester (i.e. the *VisionWatchdog* component) is appropriately connected to the service provider (i.e. the *ReconfigurationAnalysis* aspect).

```
ReconfigurationAnalysis aspect VisionSystemReconfigurationAnalysis
...
   Services
      in faultyOutput(input failingComponent: string);

   Played_Roles
      INTERNAL-EVENTS for I_WatchdogEvents ::=
         faultyOutput?(failingComponent);

...
   Triggers
      RepairImageProcessingUnit() when
         {failingComponent==["ImageProcCard"]}
         INTERNAL-EVENTS_faultyOutput?(failingComponent);
...
```

Figure 6.9. Reconfiguration triggers: activation by service invocations

Event interception. The condition is true when a certain event is intercepted in the architecture of the composite instance. Event interception is performed by means of three services, which are provided by default: *beforeEvent*, *insteadOfEvent*, and *afterEvent*. These services intercept an event or service request: *before* it is delivered to the recipients (i.e. the server instances), *after* its processing by the recipients, or *instead of* the recipients (i.e. the event is processed by this aspect, the original recipients do not receive the event). Thus, a configuration transaction can be activated as a result of: (i) a service request (*beforeEvent* and *insteadOfEvent*); or (ii) the processing of a service request (*afterEvent*). The difference is that *insteadOfEvent* replaces the execution of a service request by an alternative implementation. These services require as input the name of an event/service to intercept, *eventName*, and return a list with the parameters provided by the intercepted event/service, *eventParameters.* The mechanisms for event capturing are actually provided by the Monitoring aspect (see section 6.4.1).

For instance, the reconfiguration trigger shown in Figure 6.10 illustrates how to activate the configuration transaction *DisableVisionSystem* on the interception of the *faultyOutput* event. The main advantage of event interception with respect to service invocations is that it is done in a transparent way, without creating explicit connections to the *VisionWatchdog* component.

```
ReconfigurationAnalysis aspect VisionSystemReconfigurationAnalysis
...
   Triggers
      DisableVisionSystem() when
         {eventParams==["VideoCaptureCard"]}
         beforeEvent!("faultyOutput", out eventParams);
...
```

Figure 6.10. Reconfiguration triggers: activation by event interceptions

6.4.2.3 Configuration Transactions

A *Configuration Transaction* is a specification that describes an ordered set of domain-specific reconfiguration operations, called *Configuration Actions* (see section 6.4.3.1, page 191), which must be executed in a transactional way (i.e. all or none, see section 6.4.3.2, page 194).

In other words, the body of a configuration transaction consists of a set of configuration actions (i.e. a set of domain-specific reconfiguration operations) that, upon execution, will change the architecture of a composite instance. A configuration action is self-descriptive: its name takes the form of a reconfiguration operation and the name of the type which is manipulated. For instance, the configuration action *create-ImageProcSoftware* creates instances of the ImageProcSoftware architectural type; the configuration action *attach-Att_VCCConn_IPCSW* creates attachments defined by the type VCCConn_IPCSW (i.e. an attachment among an instance of a VCC-Conn connector type and an instance of an ImageProcSoftware component type), and so on.

These configuration actions must be *ordered*, because they describe a change process which has dependencies among its elements. For instance, we cannot create an attachment with an instance that has not been created yet. Similarly, we should not remove an architectural element if it is still connected to other elements: we must remove its attachments first.

An example of how these configuration actions are used to define a reconfiguration process is shown in Figure 6.11. This figure shows the specification of a configuration transaction called

`RepairImageProcessingUnit`. It describes how the component *ImageProcCard* must be replaced by the component *ImageProcSoftware* in case of malfunction. The transaction consists of two processes. The first one (see `REPAIRIMAGEPROCESSINGUNIT` process) obtains the references (i.e. the IDs) to the instances that are going to be affected by the reconfiguration process. Then, the second process (see `RECONF` process) performs a set of configuration actions: creates a new instance of the *ImageProcSoftware* component, attaches the new instance to the *VCC-Conn* and *IPC-Conn* connector instances, detaches the failing *ImageProcCard* instance from the previous connector instances, and finally destroys the *ImageProcCard* instance. As a result of this process, a new component instance has been created and attached to the other interacting elements, removing the failing *ImageProcCard* component instance.

```
ReconfigurationAnalysis aspect VisionSystemReconfigurationAnalysis
...
Transactions
 RepairImageProcessingUnit():
   // Configuration transaction for replacing an imageProcCard
   // component by an imageProcSoftware component
   REPAIRIMAGEPROCESSINGUNIT ::=
      // Get IDs of instances subject to changes
      // Only one instance of ImageProcCard,VCCConn and IPCConn
      // is allowed by the System type, so no iterations are needed
      oldImProcCardID = imageProcCard-list[0] -->
      VCCConnID=VCC-Conn-list[0] -->
      IPCConnID=IPC-Conn-list[0] --> RECONF;
   RECONF ::=
      create-ImageProcSoftware!(cameraPos, output newImProcID) -->
      attach-Att_VCCConn_IPCSW!(VCCConnID, newImProcID,
         output newAttID) -->
      attach-Att_IPCSW_IPCConn!(newImProcID, IPCConnID,
         output newAttID) -->
      detach-Att_VCCConn_IPC!(VCCConnID, oldImProcCardID) -->
      detach-Att_IPC_IPCConn!(oldImProcCardID, IPCConnID) -->
      destroy-ImageProcCard!(oldImProcCardID) -->
   END;
```

Figure 6.11. Example of Configuration Transactions: *RepairImageProcessingUnit*

The execution of configuration actions may be subject to failures. A configuration action may fail if the constraints defined in the System type (such as cardinalities) are violated by this action. For instance, a *create* action may fail if we try to create more instances than the maximum allowed. On the other hand, a configuration action may also fail due to technical reasons: e.g. a connection cannot achieve quiescence and cannot be removed safely.

This must be taken into account when changing a running system: if anything fails in the middle of a reconfiguration process, the resulting system could be left in an inconsistent state. For this reason, reconfiguration processes are defined as inherently transactional: such processes must be executed atomically, and they should be undoable if anything fails. This transactional support is provided in the PRISMA ADL by means of a section called *Transactions* (see Figure 6.11): every process defined in this section is executed inside a transactional context. Transactional management is entirely transparent for the System architect: this is managed by the underlying reconfiguration mechanisms (i.e. the Reconfiguration Coordination aspect, see section 6.4.3). The architect only describes: (i) the reconfiguration process (i.e. an ordered set of configuration actions), and (ii) the end of the process (by means of the `END` process, see last line from Figure 6.11).

For instance, a more complex reconfiguration transaction that relies explicitly on the use of transactions is `IncrementalStart` (see Figure 6.12). This configuration transaction is activated periodically after an unrecoverable failure of the VisionSystem. In this situation, all the subsystems (i.e. components and connectors) have been stopped and disabled for preventing additional failures. This transaction consists of three processes. The first one sets some internal variables, as the output if anything fails. The second process, `RESTART`, tries to incrementally enable each one of the subsystems that were previously stopped. This is performed by means of *replace* operations: a new instance is created and the old state is migrated, if possible. The reason is that a partial restart may solve some of temporary failures (Yurcik & Doss, 2001), even internal transient errors or external physical conditions (such as humidity, power failure, etc.). Finally, if all the elements have been replaced successfully, the `WAITING-TEST` process is executed (see Figure 6.12). This process waits for two mutually exclusive events: *validOutput* or *faultyOutput*. These events are triggered by the *VisionWatchdog* component instance after testing the behaviour of the subsystems of the *VisionSystem* (see the complete specification at appendix A.1.7.4). If all the testing is correct (i.e. the *validOutput* event is received), then the transaction is committed: this is done by calling the `END` process, which is provided by the PRISMA ADL. On the other hand, if the testing has failed (i.e. the *faultyOutput* event is received instead), the transaction is rollbacked: this is done by calling the `ROLLBACK` process. Then, all the configuration actions that had been executed are undone: the original instances are restored, and the replacements deleted.

```
ReconfigurationAnalysis aspect VisionSystemReconfigurationAnalysis
...
Transactions
   IncrementalStart(output success: boolean):
      // Tries to restart all the subsystems after a failure
```

```
INCREMENTALSTART ::=
   <success=false> --> RESTART;
RESTART ::=
   // Replaces the old instances by new, fresh ones.
   getArchElementInitializationValues!(
      videoCaptureCard-list[0], videoCardInitValues) -->
   replace-VideoCaptureCard!(videoCaptureCard-list[0],
      videoCardInitValues[0], output newID)   -->
   replace-VCC-Conn!(VCC-Conn-list[0], output newID) -->
   getArchElementInitializationValues!(visionWatchdog-list[0],
      watchdogInitValues) -->
   replace-VisionWatchdog!(visionWatchdog-list[0],
      watchdogInitValues[0], output newID) -->
   if (imageProcCard-list.Size()>0) then
      replace-ImageProcCard!(imageProcCard-list[0],
         cameraPos, output newID) -->
   foreach elem in imageProcSoftware-list do (
      replace-ImageProcSoftware!(elem,cameraPos,output newID);
   ) -->
   replace-IPC-Conn!(IPC-Conn-list[0], output newID) -->
   WAITING-TEST;
WAITING-TEST ::=
   ( INTERNAL-EVENTS_validOutput?() --> <success=true>END )
+  ( INTERNAL-EVENTS_faultyOutput?(failingID) --> ROLLBACK);
```

Figure 6.12. Example of Configuration Transactions: *IncrementalStart*

The Reconfiguration Analysis aspect is provided by default with auxiliary variables and services, to make easy the specification of reconfiguration processes. These are automatically generated for the given domain (i.e. the architecture to reconfigure). On the one hand, predefined attributes are provided for getting different kinds of information (see Figure 6.13):

- `systemID` is a read-only attribute that provides the ID of the System instance being reconfigured. This is useful for notifying external systems about reconfiguration actions performed internally.

- `<ArchitecturalElementType>-list`. For each architectural type available in the architecture being managed, an attribute named *<ArchitecturalElementType>-list* is provided. This attribute is a read-only list which contains references (i.e. the IDs) to the instances of the type named *<ArchitecturalElementType>*. This is useful for obtaining the elements that are instantiated in a System architecture, since their IDs are required for performing reconfiguration actions (e.g. see Figure 6.12). The values of these lists are provided by means of derivation rules; that is, their values are actually provided by low-level reconfiguration mechanisms (i.e. the Monitoring Aspect). However, this is transparently provided to the System architect.

```
ReconfigurationAnalysis Aspect VisionSystemAnalysisServices

   Attributes
      Derived
      // Attributes to query the current configuration (read-only)
         systemID: string, derivation:
            getArchElementInstances("self", output list[0]);
         videoCaptureCard-list : list, derivation:
            getInstances-videoCaptureCard(output
               videoCaptureCard-list);
         imageProcCard-list : list, derivation:
            getInstances-imageProcCard(output imageProcCard-list);
         imageProcSoftware-list: list, derivation:
            getInstances-imageProcSoftware(output
               imageProcSoftware-list);
         visionWatchdog-list: list, derivation:
            getInstances-visionWatchdog(output visionWatchdog-list);
         VCC-Conn-list: list, derivation:
            getInstances-VCC-Conn(output VCC-Conn-list);
         IPC-Conn-list: list, derivation:
            getInstances-IPC-Conn(output IPC-Conn-list);
...
```

Figure 6.13. Example of auxiliary attributes provided by the Rec.Analysis aspect

The services that can be used in a Reconfiguration Analysis aspect for defining reconfiguration triggers and configuration transactions are the following:

- The domain-specific reconfiguration services (i.e. configuration actions) provided by the Reconfiguration Coordination aspect. These services allow us to change the current architecture in a safe way. For instance, the service `create-ImageProcSoftware` creates a new instance of the ImageProcSoftware type, whereas the service `attach-Att_VCCConn_IPCSW` attaches an instance of a VCC-Conn connector to an instance of an ImageProcessingSoftware component.
- All the services provided by the Monitoring aspect, which allow us to introspect the current state of the managed architecture or intercept events. For instance, the service `AttachedElements` may be useful for getting the elements that are interacting with a given instance and which will be affected by the reconfiguration of such instance.
- Additionally, the services `StartElement` and `StopElement` provided by the ReconfigurationEffector aspect. These services may be useful for selectively disabling conflicting elements temporarily until a certain event is received. For instance, selective stoppings have been used by the *DisableVisionSystem* configuration transaction (see its specification in appendix A.2.3.1) for disabling misbehaving component instances. Later, when certain conditions apply (in this

case, some time has passed), a different configuration transaction (i.e. *IncrementalStart*) tries to restart the elements of the architecture.

6.4.2.4 Adding Inference Mechanisms

The *Reconfiguration Analysis* aspect defines a placeholder where reconfiguration decision mechanisms are placed. However, the decision of which inference engine use to activate and select reconfiguration actions is left to the user. We have used the PRISMA ADL to define Event-Condition-Action (ECA) policies. These policies are expressive enough to describe how a composite component should react in presence of certain events. In fact, it is the common approach used in autonomic computing approaches (Huebscher & McCann, 2008). In our approach, these policies are defined by the system architect at design-time, although they can be changed at runtime by using reflective dynamic evolution mechanisms, as described in section Chapter 7.

Other, more sophisticated, inference engines may be used for activating and selecting reconfiguration plans. For instance, Artificial Neural Networks can be used for classifying inputs from the architecture (e.g. performance, events, etc.) into reconfiguration plans, in a similar way as robot controllers are generated (Santos et al., 2001) (Buason et al., 2005). Another option is to use Inductive Systems to select a reconfiguration plan depending on the information gathered in a knowledge-base from previous reconfigurations or actions taken by the user.

However, the most challenging is to build reconfiguration plans at runtime. This is an active area of research which is out of the scope of this thesis and is left to future work. A promising work on the topic is the development of *Digital Evolution* (McKinley et al., 2008), where the dynamic generation of state diagrams is explored by means of evolutionary techniques. However, it is still in an early research phase. Another promising approach is the automatic synthesis of component configurations from high-level goals (Sykes et al., 2008). All of these mechanisms could perfectly be encapsulated inside the ReconfigurationAnalysis aspect using the expressive power of the PRISMA ADL.

6.4.3 The Reconfiguration Coordination Aspect

This aspect performs two roles. On the one hand, the Reconfiguration Coordination aspect provides the specific reconfiguration services that are available according to the System type that the aspect manages. These reconfiguration services are domain-specific (i.e. architecture-specific), and are called *configuration actions*. On the other hand, the Reconfiguration Coordination aspect is responsible for driving the successful execution of the reconfiguration plans triggered by the Reconfiguration Analysis aspect. It ensures that these plans are performed transactionally (all or none), and that the current state of the architecture is preserved.

These roles are implemented in two different parts: one defines the domain-specific reconfiguration behaviour (i.e. the configuration actions), and the other defines the domain-independent reconfiguration behaviour (i.e. the transactional management of reconfigurations and the generic reconfiguration services). This separation is due to its nature: the domain-specific behaviour is automatically generated for each System type, and the domain-independent behaviour is provided as part of the PRISMA metamodel. These parts are linked to each other by means of inheritance mechanisms: domain-independent behaviour is provided by a base aspect called *BaseReconfigurationCoordination*, and domain-specific behaviour is provided through the specialization of the base aspect. The specialized aspect is named following this pattern: `<SystemTypeName>ReconfigurationServices`. For instance, the specialized Reconfiguration Coordination aspect for the VisionSystem System type is called VisionSystemReconfigurationServices. This specialized aspect internally uses the behaviour that is inherited from its parent (i.e. the *BaseReconfigurationCoordination* aspect). Both aspects are described in detail next. Their complete specification is provided in appendix A.2.4.

6.4.3.1 Domain-Specific Reconfiguration Services

The Reconfiguration Coordination aspect is the provider of reconfiguration services: it provides a set of domain-specific reconfiguration services to change the architecture of a System instance at runtime. These are called *configuration actions*, and are automatically generated by the Dynamic PRISMA model compiler.

A *Configuration Action* is basically an architectural reconfiguration operation: an operation which, on execution, may change the architecture of a System instance, either by adding or removing elements, or by adding or removing connections among elements. This does not differ on the reconfiguration operations that other works have proposed (Bradbury et al., 2004). However, *configuration actions are domain-specific*: the set of available reconfiguration operations is defined in terms of the operations that are allowed by the architectural type of the instance being reconfigured.

This is a first step for guaranteeing that the changes performed are **type-conformant**: only instances of allowed types can be created, and they can only be attached to other elements while satisfying the communication patterns defined in the architectural type. In this way, since only previously defined (and validated) communication patterns can be used, the possibility of performing invalid configurations at runtime is reduced.

There are seven kinds of configuration actions available. These are defined in terms of the specific elements defined in the System type being reconfigured: architectural element types (i.e. component and connector types), attachment types (i.e. interaction patterns among component/connector types) and binding types (i.e. interaction patterns among architectural types and ports).

The different kinds of configuration actions are the following:

- `getInstances-<ArchitecturalTypeName>`: returns a list with the IDs of the instances of a certain type.
- `create-<ArchitecturalTypeName>`: for creating new instances of components/connectors of a certain type. This service requires a set of initialization parameters[30] (determined by the type to instantiate) and returns the ID of the new instance.
- `destroy-<ArchitecturalTypeName>`: for destroying instances of components/connectors of a certain type. This service requires the ID of the instance to destroy.
- `replace-<ArchitecturalTypeName>`: for replacing an old instance with a new instance of the same type, performing state migration if possible. This is useful for recovering/updating old instances. It allows us to provide different initialization parameters to the new instance.
- `attach-<AttachmentTypeName>`: for attaching two instances as defined in <attachmentTypeName>. This action only requires the IDs of the instances to attach.
- `detach-<AttachmentTypeName>`: for detaching two instances. This action requires the IDs of the instances to detach.
- `bind-<BindingTypeName>`: for binding an instance to an external port, as defined in <bindingTypeName>. This action only requires the ID of the instance to bind (the port name is defined in <bindingTypeName>).

[30] Generally, in order to instantiate a type, a set of initialization values may be required. For instance, the VideoCaptureCard component requires the initial frame rate value, and the ImageProcCard component requires the position of the camera for adjusting the captured images. For this reason, reconfiguration specifications should also provide the required initialization values when creating new instances or replacing existing ones.

- `unbind-<BindingTypeName>`: for removing a binding of an instance or port. This action only requires the ID of the instance to unbind.

In this way, the use of (domain-specific) configuration actions avoids the possibility of instantiating non-allowed types, removing critical instances, or establishing invalid connections among architectural elements. In other words, non-valid configuration actions cannot be used because they are simply not provided.

For instance, Figure 6.14 shows the set of domain-specific configuration actions that are available for reconfiguring*VisionSystem* instances:

```
ReconfigurationCoordination Aspect VisionSystemReconfigurationServices
   using I_VisionSystemReconfigurationServices
...

Services
   // *** DOMAIN-SPECIFIC RECONFIGURATION SERVICES ***
   //*** Create- Services ***
   in create-ImageProcCard(cameraPosition: string,
         output newInstanceID:string);
   in create-ImageProcSoftware(cameraPosition: string,
         output newInstanceID: string);

   //*** Destroy- Services ***
   in destroy-imageProcCard(instanceID : string);
   in destroy-imageProcSoftware(instanceID : string);

   //*** Replace- Services ***
   in replace-VideoCaptureCard(oldInstanceID: string,
         frameRate: natural, output newInstanceID : string);
   in replace-ImageProcCard(oldInstanceID: string,
         cameraPosition: string, output newInstanceID : string);
   in replace-ImageProcSoftware(oldInstanceID: string,
         cameraPosition: string, output newInstanceID : string);
   in replace-VisionWatchdog(oldInstanceID: string,
         timeout: natural, output newInstanceID : string);
   in replace-VCC-Conn(oldInstanceID: string,
         output newInstanceID : string);
   in replace-IPC-Conn(oldInstanceID: string,
         output newInstanceID : string);

   //*** Attach- Services ***
   in attach-Att_VCCConn_IPC(VCCConn-ID : string, IPC-ID: string);
   in attach-Att_VCCConn_IPCSW(VCCConn-ID:string,IPCSW-ID:string);
   in attach-Att_IPC_IPCConn(IPC-ID : string, IPCConn-ID: string);
   in attach-Att_IPCSW_IPCConn(IPCSW-ID:string, IPCConn-ID:string);

   //*** Detach- Services ***
   in detach-Att_VCCConn_IPC(VCCConn-ID: string, IPC-ID: string);
   in detach-Att_IPC_IPCConn(IPC-ID: string, IPCConn-ID: string);
   in detach-Att_VCCConn_IPCSW(VCCConn-ID:string,IPCSW-ID:string);
   in detach-Att_IPCSW_IPCConn(IPC-ID: string,IPCConn-ID: string);
```

```
    //*** Bind- Services ***
    // NONE
    //*** Unbind- Services ***
    // NONE
...
```

Figure 6.14. Example of domain-specific reconfiguration services

Note in the example above that not all the kinds of configuration actions are available for reconfiguring a VisionSystem architecture. This is due to the constraints defined by *VisionSystem* type (see the complete specification of the type in section A.1.4.1). For instance, there are not: (i) bind/unbind actions, (ii) create/destroy actions for *VideoCaptureCard* and *VisionWatchdog* components, or for *VCC-Conn* and *IPC-Conn* connectors, and (iii) attach/detach actions among *VideoCaptureCard* or *VisionWatchdog* instances and *VCC-Conn* instances. This is due to the cardinalities defined in the VisionSystem type.

Configuration actions do not only constrain the kinds of reconfiguration operations that can be performed, but also take into account the additional constraints defined in the System type, such as the minimum and maximum cardinalities. If a configuration action violates the constraints defined in the System type (e.g. the maximum number of instances allowed of a certain type), then the operation fails and is aborted (this checking is performed by the Reconfiguration Coordination aspect, which will be described later).

Cardinality constraints are checked at runtime, except the special case of "1..1" cardinalities, which are evaluated at compile-time. When the cardinality of an architectural element is "1..1", i.e. only an instance of this element can exist at runtime –a singleton element-, then no create/destroy configuration actions are provided for this architectural element. Except from the first instance created at the instantiation of a System, no new instances can be created, and this instance cannot be removed from the system. This also applies for attachment and binding types with a "1..1" cardinality. This can be observed in the available configuration actions of the example above (Figure 6.14). For instance, the cardinality of *VideoCaptureCard* and *VisionWatchdog* components cannot vary (i.e. their cardinality is "1..1"). Then, after the initial VisionSystem instantiation, no new instances of these components can be created or destroyed. Thus, the only alternative to modify the existing instances is by means of replace actions.

6.4.3.2 Transactional Management of Reconfiguration Plans

The Reconfiguration Coordination aspect is actually the responsible for executing reconfiguration plans (i.e. ordered sets of domain-specific reconfiguration operations). Since these reconfiguration plans are meant to change a running System instance, it must be guaranteed that their execution do not break the running system.

Reconfiguration plans clearly exhibit a transactional behaviour: their reliability depends on the same properties that Jim Gray and Andreas Reuter defined for describing reliable transactional systems (Gray & Reuter, 1993). These properties, also known as *ACID* properties, are the following:

- **Atomicity**. A reconfiguration plan must be processed as a single unit: all the required configuration actions must be executed successfully or the entire plan undone. Otherwise, a partially executed reconfiguration plan would leave the architecture in an inconsistent state. In case a reconfiguration fails, the system is left unchanged.
- **Consistency**. The execution of a reconfiguration plan must leave the architecture of the system in a consistent state. That is, in the resulting architecture: (i) all the required client interfaces are bound to corresponding server interfaces, and (ii) all the architectural elements temporarily stopped during the reconfiguration have been restarted.
- **Isolation**. The intermediate results of a reconfiguration plan should not be accessible until the reconfiguration finishes. That is, the resource being changed (i.e. the architecture) should be isolated from the other elements that perform *reads* or *writes* on this resource. In this context, the elements that *read* (or use) the architecture of a System instance are its architectural elements and ports: they interact following the topology defined by the architecture. This means that, during the processing of a reconfiguration plan, all the elements subject to changes must be isolated from the others. On the other hand, the elements that *write/modify* the architecture are other reconfiguration plans. This means that, during the processing of a reconfiguration plan, other reconfiguration plans cannot be concurrently executed on the same architecture or System instance. That is, reconfiguration plans must be sequentially executed to avoid architecture inconsistencies.
- **Durability**. This property refers to the persistence over time of the changes performed by a transaction. This is required to guarantee that, in case any kind of system failure occurs (either hardware or software), the recovered system keeps the reconfigurations performed.

In order to make explicit the transactional character of reconfiguration plans, we decided to use the term *configuration transaction*. Thus, a *configuration* *transaction* is defined as an ordered set of domain-specific reconfiguration operations (i.e. configuration actions) that are executed transactionally, i.e. satisfying the ACID properties. The transactional support for reconfigurations is provided by the Reconfiguration Coordination aspect.

A configuration transaction is delimited by two operations: *BeginConfigurationTransaction* and *EndConfigurationTransaction*. These operations delimit the beginning and the end (i.e. the commit) of a configuration transaction, respectively. During the execution of a configuration transaction, auxiliary information is kept for later undoing the changes if an error occurs. If the transaction finishes successfully (i.e. the *EndConfigurationTransaction* is executed), this information is simply removed and the new changes confirmed and persisted. Otherwise, if any configuration action fails or triggers an exception, a service called *RollbackConfigurationTransaction* is implicitly executed, which uses the previously stored auxiliary information to undo the architectural changes. This service can also be explicitly invoked to manually abort a reconfiguration transaction.

Next, the details about transactional management are described.

BeginConfigurationTransaction

The execution of this service performs the following actions. First, it prevents other configuration transactions to be started while there is another being executed, in order to guarantee the *isolation* of configuration transactions. This is performed by means of the attribute *transState*, which keeps the state of the currently processed transaction (if any). This attribute is set to `ACTIVE` if a transaction is being processed, or to `COMMITTED/ROLLBACKED` if a transaction has finished. Thus, the precondition shown in Figure 6.15 verifies that no other transaction is being processed. If this precondition is satisfied, then the *BeginConfigurationTransaction* service can be executed, and then a new configuration transaction started. Then, the attribute *transState* is set to `ACTIVE`, which in turn will prevent new configuration transactions to be started.

Next, this service initializes the auxiliary structures that will be used to store the required information to undo the configuration transaction (see *Services* section, Figure 6.15). These structures contain: the IDs of the architectural elements and connections created by the transaction (stored in the attributes called *archElementsCreated* and *connectionsCreated*, respectively); and the IDs of the architectural elements and connections that should be destroyed if the transaction is committed (stored in the attributes called *archElementsToDestroy* and *connectionsToRemove*, respectively).

Finally, the *BeginConfigurationService* creates a new transactional context for encapsulating the reconfiguration process and preserve the isolation of the elements that will be changed (i.e. the architecture). This functionality is provided by the PRISMANET middleware, and is accessible by means of an external function called *NewTransactionalContext*. This function returns the ID of the newly created transactional context, which is stored in the attribute *transactionID* (see *Transactions* section, Figure 6.15).

The concept of *transactional context* represents the limits of a transaction and guarantees the isolation of the resources that are involved in the transaction (i.e. that are subject to change). A transactional context protects the attributes of the aspect which has initiated the transaction from those services and aspects that are outside this transactional context. Thus, only the services that are executed by the transaction can read or modify these attributes, thus guaranteeing that the intermediate results of the transaction are hidden to other elements. In addition, the limits of the transactional context are not only confined to the initial aspect or architectural element. The boundaries of a transactional context are dynamically extended to: (i) include the results of nested transactions, or (ii) support distributed transactions. Further details about the implementation of the transactional support in PRISMANET can be found in (Millán-Belda, 2006, pp. 129-142) and in Ambient-PRISMA (Ali, 2008, pp. 301-310).

```
ReconfigurationCoordination Aspect
...
   Attributes
   Variable
      transactionID: string;
      transState: string = "COMMITTED"; // Default value
      archElementsCreated : list;
      archElementsToDestroy : list;
      connectionsToRemove : list;
      connectionsCreated : list;
   Derived
      ArchElemList : list,
         derivation: getArchElementInstances("*", ArchElemList);
      ConnectionsList : list,
         derivation: getConnectionsOfArchElem("*",ConnectionsList);
...
   Preconditions
      BeginConfigurationTransaction()
         if ( transState="COMMITTED" or transState<>"ROLLBACKED");
...
   Services
   // *** Transaction Management ***
      in BeginConfigurationTransaction()
         Valuations
            [BeginConfigurationTransaction()]
            transState = "ACTIVE";
            archElementsCreated = new list[];
            archElementsToDestroy = new list[];
            connectionsToRemove = new list[];
            connectionsCreated = new list[];
...
   External Functions
      NewTransactionalContext(output transactionID: string);
         // Creates a new transactional context
...
   Transactions
   BeginConfigurationTransaction():
      BEGINCONFIG::=
         NewTransactionalContext(output transactionID) --> END;
...
```

Figure 6.15. PRISMA specification of BeginConfigurationTransaction

EndConfigurationTransaction

This service is executed to define the end of a configuration transaction, and then, to commit the changes performed on the architecture. The precondition to execute this service is that a transaction is being processed (i.e. the attribute *transState* has the value `ACTIVE`), and that this transaction has been successfully executed until now (i.e. the attribute *transState* do not have the value `ROLLBACK`, which is set when an error occurs). This is evaluated in the preconditions section, shown in Figure 6.16.

The execution of the *EndConfigurationTransaction* service performs the following actions. First it verifies the *consistency* of the resulting architecture (see the `CHECK` process in the transactions section in Figure 6.16). That is, all the client ports are bound to server ports. This checking is performed by the service *CheckArchitectureConsistency* (see Figure 6.17). This service checks that each port with a "requires" role (i.e. a client behaviour, it is a service requester) is correctly bound to a port with the same interface and a "provides" role (i.e. a server behaviour, it is a service provider). If this checking fails (i.e. the resulting architecture is not consistent), then the configuration transaction is rollbacked (i.e. the *RollbackConfigurationTransaction* service is executed), leaving the architecture in the previous state.

Next, the architectural changes are committed, i.e. made permanent. This is done in the `COMMIT` process (see the transactions section in Figure 6.16), which:

- Confirms the removals that have been performed by the configuration transaction. The IDs of the elements to remove, i.e. connections and architectural elements, are stored in the attributes *connectionsToRemove* and *archElementsToDestroy*. Then, these elements are destroyed, by means of the services *Disconnect* and *DestroyInstance*.

Once the elements are removed/destroyed, the operation cannot be undone, since their internal state is lost.

- Starts the architectural elements and connections that have been created by the transaction. The elements to start are obtained from the attributes *archElementsCreated* and *connectionsCreated*. First the new connections are enabled, and then the architectural elements, which may use these connections.
- Restarts the elements that were stopped temporarily by the configuration transaction. These elements are those that have the status "passivated". Passivating allows us to partially stop an element with respect other elements, avoiding that this element starts new interactions with the others.
- Saves the new configuration in disk, in order to make the changes permanent. If the System instance is restarted, its architecture will include the reconfigurations performed previously. This is performed by: (i) obtaining the resulting architecture specification (which is a XML description provided by the service *getConfigurationSpecification*), and (ii) saving it in the filesystem (by means of the external function *SavePRISMASpec*).

Finally, the configuration transaction is finished: (i) the transactional context is finished, through the external function *FinishTransactionalContext* (see the `FINISH` process in Figure 6.16); and (ii) the attribute *transState* is set to `COMMITTED`, to allow new configuration transactions to be processed (see the Valuations section in Figure 6.16).

```
ReconfigurationCoordination Aspect
...
   Preconditions
      EndConfigurationTransaction() if (transState="ACTIVE" );
         // EndConfiguration is only allowed if a transaction exists
...
   Services
      in EndConfigurationTransaction()
         Valuations
            [EndConfigurationTransaction()]
            transState="COMMITTED";
...
   External Functions
      FinishTransactionalContext(transactionID: string);
         // Destroys a transactional context and makes changes
         // permanent
      SavePRISMASpec(name: string, specification: string);
         // Saves a PRISMA specification (a type or a configuration)
         // in the filesystem
...
```

```
   Transactions
   EndConfigurationTransaction():
      CHECK::=
         CheckArchitectureConsistency(output isConsistent) -->
         if { isConsistent=true AND transState="ACTIVE" }
            then COMMIT
            else RollbackConfigurationTransaction();
      COMMIT::=
         foreach connID in connectionsToRemove do
            ( Disconnect!(connID) )          -->
         foreach archElemID in archElementsToDestroy do
            ( DestroyInstance!(archElemID) ) -->
         foreach connID in connectionsCreated do
            ( StartElement!(connID) )        -->
         foreach archElemID in archElementsCreated do
            ( StartElement!(archElemID) )    -->
         getElementsOfStatus("passive", passivatedElements) -->
         foreach stoppedElem in passivatedElements do
            (StartElement!( stoppedElem ) )  -->
         getConfigurationSpecification!(output newConfig) -->
         SavePRISMASpec(systemID, newConfig) --> FINISH;
      FINISH::= FinishTransactionalContext(transactionID) --> END;
...
```

Figure 6.16. PRISMA specification of EndConfigurationTransaction

```
// Checks that all the required ports are bound to provided ports
CheckArchitectureConsistency(output isConsistent: boolean):
   CHECKCONSISTENCY::=
      foreach archElemID in ArchElemList do (
         getArchElementProperties!(archElemID,_, output portsList)
         -->
         foreach port in portsList do (
            getPortProperties!(archElemID, port, output isProvided,
               output isRequired,_, output connectionList) -->
            if { isRequired=true}
               then (
                  if {connectionList.Size()=0}
                  // A disconnected port has been found
                  then ( <isConsistent:=false> --> END )
               )
               else 0
         )
      ) -->
      <isConsistent:=true> --> END;
```

Figure 6.17. PRISMA specification of CheckArchitectureConsistency

RollbackConfigurationTransaction

This service is executed to abort a configuration transaction, and then, to undo the changes performed on the architecture. The precondition to execute this service is that a transaction is being processed (i.e. the attribute *transState* has the value `ACTIVE`), or that this transaction has been failed (i.e. the attribute *transState* has set to `ROLLBACK`). This is evaluated in the preconditions section, shown in Figure 6.18. This service can be executed explicitly (i.e. by a configuration transaction) if some desired conditions do not hold; or implicitly, if a configuration action fails unexpectedly.

The execution of the *RollbackConfigurationTransaction* service rollbacks the reconfiguration process. This is done in the `ROLLBACK` process (see the transactions section in Figure 6.16), which:

- Undoes the creations that have been performed by the configuration transaction. The IDs of the elements created, i.e. connections and architectural elements, are obtained from the attributes *connectionsCreated* and *archElementsCreated*. Then, these elements are destroyed, by means of the services *Disconnect* and *DestroyInstance*.
- Undoes the removals that have been performed by the configuration transaction. Since removals cannot be undone, removals are not actually executed until the confirmation of the transaction. If the transaction is not confirmed and must be undone, simply these removals are not performed, and the elements to remove are restarted again. The elements to start are obtained from the attributes *archElementsToDestroy* and *connectionsToRemove*. First the connections are enabled, and then the architectural elements, which may use these connections.
- Restarts the elements that were stopped temporarily by the configuration transaction. These elements are those that have the status "passivated". Passivating allows us to partially stop an element with respect other elements, avoiding that this element starts new interactions with the others.

Finally, the current configuration transaction is finished: (i) the transactional context is finished, through the external function *FinishTransactionalContext* (see the `FINISH` process in Figure 6.18); and (ii) the attribute *transState* is set to `ROLLBACKED`, to allow new configuration transactions to be processed (see the Valuations section in Figure 6.18).

```
ReconfigurationCoordination Aspect
...
   Preconditions
      RollbackConfigurationTransaction()
         if (transState=”ACTIVE” or transState=”ROLLBACK”);
         // Rollback only when a transaction exists
...
   Services
      in RollbackConfigurationTransaction();
         Valuations
```

```
            [RollbackConfigurationTransaction()]
            transState="ROLLBACKED";
...
   Transactions
   RollbackConfigurationTransaction():
      ROLLBACK::=
         foreach connID in connectionsCreated do
            ( Disconnect!(connID) )          -->
         foreach archElemID in archElementsCreated do
            ( DestroyInstance!(archElemID) ) -->
         foreach archElemID in archElementsToDestroy do
            ( StartElement!(archElemID) )    -->
         foreach connID in connectionsToRemove do
            ( StartElement!(connID) )        -->
         getElementsOfStatus("passive", passivatedElements) -->
         foreach stoppedElem in passivatedElements do
            (StartElement!( stoppedElem) )   --> FINISH;
      FINISH::= FinishTransactionalContext(transactionID) --> END;
...
```

Figure 6.18. PRISMA specification of RollbackConfigurationTransaction

Finally, we summarize here how the Reconfiguration Coordination aspect guarantees the ACID properties for configuration transactions:

- **Atomicity** of reconfigurations is guaranteed by means of the internal attributes of the aspect, which keep "a journal" of the reconfigurations realised, and the rollback of these changes if any intermediate operation fails.
- The architectural **consistency** after a configuration transaction is evaluated by the service *EndConfigurationTransaction*. If the resulting architecture is not consistent, the transaction is undone.
- **Isolation** is performed together by preconditions and transactional contexts. The preconditions of *BeginConfigurationTransaction* and *EndConfigurationTransaction* prevent other configuration transactions to be executed while one is being processed. And, transactional contexts protect the architecture being changed to be accessed by services not included in the current transaction.
- **Durability** of the reconfigurations is performed by the service *EndConfigurationTransaction*. When a transaction is committed, the resulting configuration is saved in disk for enabling recovering it in the future.

6.4.3.3 Generic Reconfiguration Services

The Reconfiguration Coordination aspect provides configuration actions as a way of guaranteeing that only type-conformant reconfigurations are performed. These configuration actions are type-dependent, i.e. they are specific for a System type. However, internally these configuration actions invoke *generic reconfiguration services*, i.e. services that provide domain-independent reconfiguration behaviour (see the protocol section of the VisionSystemReconfigurationServices aspect, Figure 6.19).

```
ReconfigurationCoordination Aspect VisionSystemReconfigurationServices
   using I_VisionSystemReconfigurationServices
...
   Protocol
   VISIONSYSTEMRECONFIGURATION ::= begin() --> WAITING;
   WAITING ::=
      RECONFPLANS.BeginConfigurationTransaction?() --> RECONFIG;
   RECONFIG ::=
   (  // *** Provided Reconfiguration Services ***
         RECONFPLANS.create-imageProcCard?(cameraPosition, newID)
         --> CreateArchElem("ImageProcCard",
               new list[cameraPosition], newID)
      +  RECONFPLANS.destroy-imageProcCard?(instanceID)
         --> DestroyArchElem("ImageProcCard",instanceID)
      +  RECONFPLANS.replace-imageProcCard?(oldInstanceID,
            cameraPosition, newInstanceID)
         --> ReplaceArchElem(oldInstanceID,"ImageProcCard",
            new list[cameraPosition], newInstanceID)
      +  RECONFPLANS.attach-Att_VCCConn_IPC?(VCC-ConnID, IPC-ID)
         --> CreateAttachment("Att_VCCConn_IPC", VCC-ConnID,
            IPC-ID, newAttID)
      +  RECONFPLANS.detach-Att_VCCConn_IPC?(VCC-ConnID, IPC-ID)
         --> DestroyAttachment("Att_VCCConn_IPC", VCC-ConnID,
            IPC-ID)
      +  RECONFPLANS.create-imageProcSoftware?(cameraPos, newID)
         --> CreateArchElem("ImageProcSoftware",
            new list[cameraPos], newID)
      +  RECONFPLANS.destroy-imageProcSoftware?(instanceID)
         --> DestroyArchElem("ImageProcSoftware",instanceID)
      +  RECONFPLANS.replace-imageProcSoftware?(oldInstanceID,
            cameraPosition, newInstanceID)
         --> ReplaceArchElem(oldInstanceID, "ImageProcSoftware",
               new list[cameraPosition], newInstanceID)
      +  RECONFPLANS.attach-Att_VCCConn_IPCSW?(VCC-ConnID,IPCSW-ID)
         --> CreateAttachment("Att_VCCConn_IPCSW", VCC-ConnID,
            IPCSW-ID, newAttID)
      +  RECONFPLANS.detach-Att_VCCConn_IPCSW?(VCC-ConnID,IPCSW-ID)
         --> DestroyAttachment("Att_VCCConn_IPCSW", VCC-ConnID,
            IPCSW-ID)
            ...
            ... // Rest of available Reconfiguration Services
            ...
      ) --> RECONFIG
...
End ReconfigurationCoordination Aspect
      VisionSystemReconfigurationServices;
```

Figure 6.19. Protocol section of the VisionSystemReconfigurationServices aspect

Generic reconfiguration services are defined by the Reconfiguration Coordination base aspect, and describe the application-independent actions that must be performed for each kind of reconfiguration operation (i.e. create instances, disconnect instances, replace instances, etc.). Generally, each generic reconfiguration service performs the following steps:

- Evaluation of preconditions, to check if the parameters provided are correct and if the constraints defined in the System type are satisfied (e.g. max and min cardinalities).
- Stopping of elements (mainly in removals): the running transactions of the elements affected by a reconfiguration action are finished in a safe way, to avoid disruptions to the rest of the system. Usually this implies passivating the adjacent elements.
- Apply the low-level changes (e.g. destroying the required instance and its connections).
- Check the consistence of the transactional context, to evaluate if any error has been raised unexpectedly. In this case, the transaction is rollbacked automatically.
- If the service has been finished successfully, the ID of the element created/removed is saved, in order to later undo the operation if anything fails.

Next, the specification of these generic reconfiguration services in PRISMA AOADL is presented.

CreateArchitecturalElement

This service creates a new instance of an architectural type. It requires the name of the architectural type to instantiate, *typeName*, and the initialization values required by such type, *params*. It returns the ID of the instance created, if the operation is executed successfully. The specification of this service is shown in Figure 6.20.

```
ReconfigurationCoordination Aspect
...
   Preconditions
   // Unused parameters are assigned to the "void" variable,
   // represented as '_', for clarity purposes
   CreateArchElem(typeName, params, newID) if
      (  getArchElementInstances!(typeName, output instances) AND
         META-TYPEDESCRIPTION_getArchTypeProperties!
            (typeName, _, _, _, output AEmaxCard) AND
         instances.Size() < AEmaxCard        );
...
```

```
   Transactions
   CreateArchElem(typeName, params, output newID):
      CREATE::= CreateInstance!(typeName, params, output newID) -->
                  CHECK;
      CHECK::= CheckConsistence(output transState) -->
         if {transState="ROLLBACK"}
         then RollbackConfigurationTransaction()
         else archElementsCreated.add(newID) --> END;
...
```

Figure 6.20. PRISMA specification of CreateArchElem

The preconditions to execute this service are that: (i) *TypeName* is a valid architectural type (i.e. it is a type defined in the System type), and (ii) the number of existing instances of *typeName* is lower than the maximum cardinality, *AEmaxCard*, defined for such type in the System type.

The information required from the System type is obtained by means of the meta-level, which contains the description of the System type. Communications with the meta-level are performed by means of the played_role `META-TYPEDESCRIPTION`. This played_role provides services to introspect the System type, such as: *getArchElementTypes*, *getAttachmentTypes*, *getBindingTypes*, *getArchTypeProperties*, etc. (see the interface I_CompositeTypeDescription in appendix A.3.1). In this case, the service *getArchTypeProperties* is used to obtain the maximum cardinality defined by the System type for a certain architectural type. This service fails if the requested type, *typeName*, is not defined in the System type. Otherwise, it returns the maximum cardinality, *AEMaxCard*, which is used to evaluate the precondition.

If the preconditions are satisfied, this service creates an instance of the required type, by calling the platform-dependent service *CreateInstance*, which is actually provided by the Reconfiguration Effector.

Finally, this service verifies the consistence of the transactional context. If the service *CreateInstance* has failed, the transactional context will capture the exception and will be invalidated. The service *CheckConsistence* (an external function provided by the middleware) checks this situation. If the transactional context has been invalidated, the state of the current transaction (*transState*) is set to `ROLLBACK` and the transaction is rollbacked. Otherwise, the ID of the new instance, *newID*, is stored in the list *archElementsCreated*, to allow us undo the transaction if anything fails.

CreateAttachment and CreateBinding

These services create a new connection: an attachment or a binding, respectively. *CreateAttachment* requires the following parameters: (i) the type of the attachment to create, *attachType*; and (ii) the IDs of the architectural instances to be attached, *srcAE-ID* and *trgAE-ID*. Similarly, *CreateBinding* requires: (i) the type of the binding to create, *bindingType*; and (ii) the ID of the architectural instance which will be bound to a System port, *archElemID*. Both services return the ID of the new connection created: *attID* and *bindID*, respectively. The specification of these services is shown in Figure 6.21.

The preconditions to execute these services are quite similar: (i) the IDs of the architectural instances exist, (ii) the attachment/binding type is valid (i.e. defined in the System type), and (iii) the number of existing connections of such attachment/binding type is lower than the maximum cardinality defined in the System type.

Similarly to the service *CreateArchitecturalElement*, the information required from the System type is obtained by means of the played_role META-TYPEDESCRIPTION. In this case, the services *getAttachmentTypeProperties* and *getBindingTypeProperties* are used to obtain the maximum cardinalities allowed.

```
ReconfigurationCoordination Aspect
...
   Preconditions
   // Unused parameters are assigned to the "void" variable,
   // represented as '_', for clarity purposes
   CreateAttachment(attachType, srcAE-ID, trgAE-ID, attID) if
      (  ArchElemList.Contains(sourceAE-ID) AND
         ArchElemList.Contains(targetAE-ID) AND
         META-TYPEDESCRIPTION_getAttachmentTypeProperties!(
            attachType, _, _, _, output srcMaxCard, _, _, _,
            output trgMaxCard) AND
         getAttachedArchElems!(srcAE-ID, attachType,
            output attached2src) AND
         attached2src.Size() < srcMaxCard AND
         getAttachedArchElems!(trgAE-ID, attachType,
            output attached2trg) AND
         attached2trg.Size() < trgMaxCard          );
   CreateBinding(bindingType, archElemID, bindID) if
      (  ArchElemList.Contains(archElemID) AND
         getConnectionsByType!(bindingType, output bindings) AND
         META-TYPEDESCRIPTION_getBindingTypeProperties!
            (bindingType, _, _, _, _, output trgMaxCard) AND
         bindings.Size() < trgMaxCard);
...
   Transactions
   CreateAttachment(attachType, srcAE-ID, trgAE-ID, output attID):
      ATTACH::=
         META-TYPEDESCRIPTION_getAttachmentTypeProperties!
            (attachType, _, output srcAEport, _, _, _,
               output trgAEport, _, _) -->
         Connect!(srcAE-ID, srcAEport, trgAE-ID, trgAEport,
            output attID) --> CHECK;
      CHECK::= CheckConsistence(output transState) -->
         if {transState="ROLLBACK"}
         then RollbackConfigurationTransaction()
         else connectionsCreated.add(attID) --> END;

   CreateBinding(bindingType, archElemID, output bindID):
      BIND::=
         META-TYPEDESCRIPTION_getBindingTypeProperties!(bindingType,
               output systemPortName, _, output trgAEport, _, _) -->
         Connect!("self", systemPortName, archElemID, trgAEport,
            output bindID) --> CHECK;
      CHECK::= CheckConsistence(output transState) -->
         if {transState="ROLLBACK"}
         then RollbackConfigurationTransaction()
         else connectionsCreated.add(bindID) --> END;
...
```

Figure 6.21. PRISMA specification of CreateAttachment & CreateBinding

If the preconditions are satisfied, these services create the appropriated connections (i.e. an attachment or a binding), by calling the platform-dependent service *Connect* (which is provided by the Reconfiguration Effector). This service creates a connection among two ports of two instances. Since the name of the involved ports is provided by the attachment/binding type, first they are retrieved from the meta-level. Then, the *Connect* service is invoked. In the case of an attachment, the connection is created among the ports of two architectural instances created in the architecture of a System instance. In the case of a binding, the connection is created among a port of an architectural instance and a port of the System instance. For this reason, one of the parameters of the *Connect* service is "self", which is the ID of the System instance itself.

Finally, the last operation performed by both services is the verification of the consistence of the transactional context. This is performed by the service *CheckConsistence*: if the transactional context has been invalidated, the state of the current transaction (*transState*) will be set to ROLLBACK and the transaction undone. Otherwise, the ID of the new connection is stored in the list *connectionsCreated*, to allow later undoing the transaction if anything fails.

DestroyArchElem

This service prepares an architectural instance to be removed: it is stopped and its connections (if any) removed. It requires the ID of the instance to be destroyed, *id*, and the name of its architectural type, *typeName*. The specification of this service is shown in Figure 6.22.

The preconditions to execute this service are the following: (i) *typeName* is valid (i.e. it is an architectural type defined in the System type), (ii) *id* is an instance of *typeName*, and (iii) the number of existing instances is greater than the minimum cardinality defined in the System type (i.e. an instance cannot be removed if this implies breaking the minimum cardinality). Similarly to the service *CreateArchitecturalElement*, the information required from the System type (i.e. the minimum cardinality) is obtained by means of the played_role META-TYPEDESCRIPTION.

```
ReconfigurationCoordination Aspect
...
   Preconditions
   // Unused parameters are assigned to the "void" variable,
   // represented as '_', for clarity purposes
   DestroyArchElem(typeName, id) if
      (  getArchElementInstances!(typeName, output instances) AND
         instances.Contains(id) AND
         META-TYPEDESCRIPTION_getArchTypeProperties!
            (typeName, _, _, output AEminCard, _) AND
         instances.Size() > AEminCard);
...
   Transactions
   DestroyArchElem(typeName, id):
      STOP ::= StopElement!(id) -->
         GetStatus!(id, output status) -->
         if {status="Blocked"} then STOPCONNECTIONS else STOP;
      STOPCONNECTIONS ::=
         GetConnectionsOfArchElem!(id, output connectionList) -->
         for each conn in connectionList do ( StopElement!(conn) )
         --> REMOVECONNECTIONS;
      REMOVECONNECTIONS ::=
         for each conn in connectionList do (
            GetStatus!(conn, output status) -->
            if {status="Blocked"} then connectionsToRemove.add(conn)
            else STOPCONNECTIONS
         ) --> CHECK;
      CHECK::= CheckConsistence(output transState) -->
         if {transState="ROLLBACK"}
         then RollbackConfigurationTransaction()
         else archElementsToDestroy.add(id) --> END;
...
```

Figure 6.22. PRISMA specification of DestroyArchElem

If the preconditions are satisfied, this service prepares the instance to be removed. First, the instance is safely stopped (i.e. it is driven to a quiescent status). This is performed by executing the service *StopElement*, provided by the Reconfiguration Effector. This may require passivating the adjacent instances, to avoid that they may start new interactions with the stopped instance. Next, the connections of the stopped instance are stopped and prepared for removal. This is performed by adding the connection IDs to the list *connectionsToRemove*.

Finally, the consistence of the transactional context is verified to check if any error has occurred during the execution of the previous operations. If the transactional context is valid, then the ID of the instance to destroy is stored in the list *archElementsToDestroy*. Since the destruction of instances cannot be undone (because their internal state is lost), this service actually postpones the destruction until the transactional commit. If the configuration transaction ends successfully, then the *EndConfigurationTransaction* service will confirm (and perform) the destruction of instances. Otherwise, the *RollbackConfigurationTransaction* will restart these instances marked for removal. This allows us to "undo" the removals without losing the internal state of the elements removed.

DestroyAttachment and DestroyBinding

These services destroy a connection: an attachment or a binding, respectively. *DestroyAttachment* requires the following parameters: (i) the type of attachment to remove, *attachType*; and (ii) the IDs of the architectural instances to be detached, *srcAE-ID* and *trgAE-ID*. Similarly, *CreateBinding* requires: (i) the type of the binding to remove, *bindingType*; and (ii) the ID of the architectural instance which will be unbound from a System port, *archElemID*. The specification of these services is shown in Figure 6.23.

The preconditions to execute these services are similar to *CreateAttachment* and *CreateBinding*, respectively: (i) the IDs of the architectural instances to disconnect exist, (ii) the attachment/binding type is valid (i.e. defined in the System type), and (iii) the number of existing connections of such attachment/binding type is *greater* than the minimum cardinality defined in the System type (i.e. a connection cannot be removed if this implies breaking the minimum cardinality). Similarly as the other services, the information required from the System type (i.e. the minimum cardinality) is obtained by means of the played_role META-TYPEDESCRIPTION.

```
ReconfigurationCoordination Aspect
...
   Preconditions
   // Unused parameters are assigned to the "void" variable,
   // represented as '_', for clarity purposes
   DestroyAttachment(attachType, srcAE-ID, trgAE-ID) if
      (  ArchElemList.Contains(sourceAE-ID) AND
         ArchElemList.Contains(targetAE-ID) AND
         META-TYPEDESCRIPTION_getAttachmentTypeProperties!(
            attachType, _, _, output srcMinCard, _, _, _,
            output trgMinCard, _) AND
         getConnectionsByType!(attachType, output connections) AND
         connections.Size()>0 AND
         getAttachedArchElems!(srcAE-ID, attachType,
            output attached2src) AND
         attached2src.Size() > srcMinCard AND
         getAttachedArchElems!(trgAE-ID, attachType,
            output attached2trg) AND
         attached2trg.Size() > trgMinCard
      );
   DestroyBinding(bindingType, archElemID) if
      (  ArchElemList.Contains(archElemID) AND
         getConnectionsByType!(bindingType, output bindings) AND
         META-TYPEDESCRIPTION_getBindingTypeProperties!
            (bindingType, _, _, _, output trgMinCard, _) AND
         bindings.Size() > trgMinCard
      );
...
   Transactions
   DestroyAttachment(attachType, srcAE-ID, trgAE-ID):
      GETATTID ::= getConnectionsByType!(attachType, connList) -->
         for each conn in connList do (
            getAttachmentProperties!(conn, output archElem1,
               output archElem2) -->
            if {archElem1=srcAE-ID AND archElem2=trgAE-ID}
            then ( <attachmentID=conn> --> STOP )
            else 0
         );
      STOP ::= StopElement!(attachmentID) -->
         GetStatus!(attachmentID, output status) -->
         if {status="Blocked"} then CHECK else STOP;
      CHECK::= CheckConsistence(output transState) -->
         if {transState="ROLLBACK"}
         then RollbackConfigurationTransaction()
         else connectionsToRemove.add(attachmentID)  --> END;

   DestroyBinding(bindingType, archElemID):
      GETBIND-ID ::= getConnectionsByType!(bindingType,connList) -->
         for each conn in connList do (
            getBindingProperties!(conn, _, output AE-ID) -->
            if {AE-ID=archElemID}
            then ( <bindingID=conn> --> STOP )
            else 0
         );
      STOP ::= StopElement!(bindingID) -->
         GetStatus!(bindingID, output status) -->
         if {status="Blocked"} then CHECK else STOP;
      CHECK::= CheckConsistence(output transState) -->
         if {transState="ROLLBACK"}
         then RollbackConfigurationTransaction()
         else connectionsToRemove.add(bindingID)  --> END;
...
```

Figure 6.23. PRISMA specification of DestroyAttachment & DestroyBinding

If the preconditions are satisfied, this service prepares the connection to be removed. First, the ID of the target connection is obtained. In order to do this, this service gets all the connections of the attachment/binding type to be removed. Then, by means of introspection services, this service selects the connection that links the desired instances and keeps the connection ID.

Next, the connection is safely stopped (i.e. it is driven to a quiescent status). This is performed by executing the service *StopElement*, provided by the Reconfiguration Effector. This requires passivating the connected instances, to finish their current interactions and avoid starting new ones.

Once the connection is stopped (i.e. there are no pending interactions over it, the consistence of the transactional context is verified. If the transactional context is valid, then the ID of the stopped connection is stored in the list *connectionsToRemove*.

ReplaceArchElem

Finally, this service replaces an architectural instance by another, compatible one. *ReplaceArchElem* requires the following parameters: (i) the ID of the architectural instance to replace, *IDToReplace*; (ii) the architectural type from which the new instance will be created, *newType* (this type can be the same of *IDToReplace* or a different, compatible one); and (iii) the initialization values, *initValues*. As a result, this service returns the ID of the new instance, *newID*. The specification of this service is shown in Figure 6.24.

The preconditions to execute this service are that: (i) the ID of the architectural instance to replace exists; (ii) the architectural type from which the new instance will be created is valid (i.e. it is defined in the System type); and (iii) if *newType* is different from the type of the instance to replace, *oldType*, then after the replacement the minimum cardinality of *oldType* and the maximum cardinality of *newType* is satisfied. Similarly as other services, cardinality information is obtained by means of the played_role `META-TYPEDESCRIPTION`.

```
ReconfigurationCoordination Aspect
...
   Preconditions
   ReplaceArchElem(IDToReplace, newType, initValues, newID) if
      (  ArchElemList.Contains(IDToReplace) AND
         META-TYPEDESCRIPTION_getArchElementTypes!(output AETypes)
         AND   AETypes.Contains(newType)     AND
         typeOf(idToReplace, output oldType) AND
         (oldType=newType OR
           (getArchElementInstances!(newType, output inst1) AND
            META-TYPEDESCRIPTION_getArchTypeProperties!
               (newType, _, _, _, output maxCard) AND
            inst1.Size() < AEmaxCard AND
            getArchElementInstances!(oldType, output inst2) AND
            META-TYPEDESCRIPTION_getArchTypeProperties!
```

```
               (oldType, _, _, output minCard, _) AND
            inst2.Size() > minCard
           )
         )
      );
...
   Transactions
   ReplaceArchElem(idToReplace, newType, initValues, output newID):
      STOPOLDELEM ::= StopElement!(idToReplace) -->
         GetStatus!(idToReplace, output status) -->
         if {status="Blocked"}
         then STOPCONNECTIONS
         else STOPOLDELEM;
      STOPCONNECTIONS ::= GetConnectionsOfArchElem!(idToReplace,
             output connectionList)  -->
         for each conn in connectionList do (
            StopElement!(conn)       ) --> MIGRATE;
      MIGRATE ::=
         typeOf!(idToReplace, output oldType) -->
         IsSerializableType!(oldType, output isSerializable) -->
         // Check if old state can be migrated
         if {isSerializable=false}
         then (
            // State of old instance cannot be obtained: it is lost
            CreateInstance!(newType, initValues, output newID)
         )
         else (
            // State of old instance can be obtained:
            SerializeState!(idToReplace, output oldState) -->
            // Check if simple instance replacement is performed
            if {oldType=newType}
            then (
               CreateInstanceFromSerializedState!(oldType, oldState,
                  output newID)
            )
            else (
               // We are performing type updating
               // Check if new type can convert old state
               CanMigrateStateFromOldVersions!(newType,
                  output canMigrate) -->
               if {canMigrate=true}
               then (
                  ConvertStateFromPreviousVersion!(newType,
                     initValues, oldType, oldState,
                     output transformedState) -->
                  CreateInstanceFromSerializedState!(newType,
                     transformedState, output newID)
               )
               else (
                  // New type cannot accept old structures. Then the
                  // old state is lost
                  CreateInstance!(newType, initValues, output newID)
               )
            )
         )
         --> MIGRATEOLDCONNS;
      MIGRATEOLDCONNS ::=
         for each conn in connectionList do (
            typeOf!(conn, output connType) -->
            isBinding!(conn, output result) -->
            if {result=true} then (
               // If conn is a binding connection, recreate...
               CreateBinding(connType, newID, output newbindID) -->
               DestroyBinding(conn)
            )
            else (
               // If is an attachment connection, recreate...
               getAttachmentProperties!(conn, output archElem1,
                  output archElem2) -->
               if {archElem1=idToReplace}
               then CreateAttachment(connType, newID, archElem2,
                      output attID)
               else CreateAttachment(connType, archElem1, newID,
                      output attID) -->
               DestroyAttachment(conn)
            )
         ) --> CHECK;
      CHECK::= CheckConsistence(output transState) -->
         if {transState="ROLLBACK"}
         then RollbackConfigurationTransaction()
         else (   archElementsToDestroy.add(idToReplace) -->
                  archElementsCreated.add(newID) ) -->
         END;

...
```

Figure 6.24. PRISMA specification of ReplaceArchElem

If the preconditions are satisfied, this service replaces the instance *idToReplace*. This is performed in three steps. First the service creates a new instance from *newType*, migrates the state and connections of *idToReplace* to the new instance, and finally destroys the old instance. First of all, the instace to replace and its connections are safely stopped (i.e. they are driven to a quiescent status). This is performed by executing the service *StopElement*, provided by the Reconfiguration Effector.

Next, the old instance is replaced by a new one, migrating the old state if possible (see the `MIGRATE` process in Figure 6.24). The internal state of the old instance can be only migrated if the following three conditions hold: (i) the old instance allows us to export its internal state (i.e. it is a serializable type), (ii) *newType* provides a function to create new instances using the state previously exported, and (iii) if the type of the instance to replace and *newType* are different, then *newType* must provide a function to transform the state of the old instance to the data structures used by the new type. If these conditions are not satisfied, then the old instance is replaced by a new instance of *newType* and its internal state is lost. The migration of the old instance is performed by means of the services *SerializeState*, *CreateInstanceFromSerializedState* and *ConvertStateFromPreviousVersion*, which are provided by the Reconfiguration Effector aspect. The details about how these methods perform the migration are described in section 6.4.4.

Finally, the connections of the old instance are recreated for the new instance (see the `MIGRATEOLDCONNS` process in Figure 6.24). This is simply performed by creating a new attachment/binding (depending on the type of connection) to the new instance, and destroying the old one.

The last step is to check the consistence of the transactional context. If the transactional context is valid, then the IDs of the old instance and the new instance are stored in the lists *archElementsToDestroy* and *archElementsCreated*, respectively. That is, the destruction is postponed until the transactional commit. In case the configuration transaction fails, the old instance and its connections will be restarted, whereas the new instance and its connections will be removed. In case the configuration transaction ends successfully, then the old instance and its connections will be destroyed, and the new instance and its connections started. The final result is then achieved: an instance has been effectively replaced by another, keeping its existing connections and state (if the migration is possible).

6.4.4 The Reconfiguration Effector Aspect

This aspect *effects*, or performs, changes on the architecture it manages. It provides a set of atomic, simple reconfiguration services which execute low-level (i.e. platform-dependent) behaviour to change the structure of a running System instance. These reconfiguration services are simple because they do not take into account the status (i.e. whether the element has been previously stopped or not) and/or the relations with the adjacent architectural elements. They must be correctly coordinated to carry out a safe reconfiguration: this is performed by the Reconfiguration Coordination aspect (see section 6.4.3).

The implementation of these reconfiguration services is technology-dependent: depending on the technology selected and how the component execution model has been implemented, the dynamic updating mechanisms to use will be different. However, the importance here is not the implementation of these mechanisms, but the **identification of the minimum services required to support the reconfiguration process** without the need to include low-level details.

The specification of the Reconfiguration Effector Aspect is shown in Figure 6.25, and its services are described below.

```
ReconfigurationEffector Aspect
// ************************************************************
// Platform-dependent aspect for changing System instances at
// runtime.
//
```

```
// Here are described the public services that are provided.
// Internal behaviour is provided by low-level implementations.
// CANNOT BE CHANGED BY THE USER
// *****************************************************************

   Services
      // *** Services for Safe Stopping ***
1     in StartElement(elemID: string);
         // Reach an Active status.
2     in StopElement(elemID: string);
         // Reach a Quiescent status.
         // This may require the passivation of neighbours
3     in PassivateElement(elemToPassivate: string,
            blockedElement: string);
         // Passivates an element with respect the interactions with
         // another element

      // *** Basic services for Reconfiguration ***
4     in CreateInstance(typeName: string, initParams: list,
            output instanceID: string);
5     in DestroyInstance(instanceID: string);
6     in Connect(instance1: string, port1: string,
            instance2: string, port2: string, output connID:string);
7     in Disconnect(connectionID: string);

      // *** Auxiliar services for Mobility, Recovery and Updating
8     in IsSerializableType(typeName: string,
            output isSerializable: boolean);
9     in SerializeState(instanceID: string, output state: string);
10    in CreateInstanceFromSerializedState(typeName: string,
            serializedState: string, output instanceID: string);
11    in CanMigrateStateFromOldVersions(typeName: string,
            output canMigrate: boolean);
12    in ConvertStateFromPreviousVersion(typeName: string,
            oldType: string, oldState: string,
            newRequiredValues:list, output transformedState:string);

End ReconfigurationEffector Aspect;
```

Figure 6.25. Services of the Reconfiguration Effector aspect

6.4.4.1 Services for Safe Stopping

The Reconfiguration Effector provides three services for managing the execution and safe stopping (i.e. quiescence) of the elements of a System instance (see services 1-3 of Figure 6.25):

- `StartElement`. This service activates the execution of an element, *elemID*. *ElemID* is a reference to an instance, a link (i.e. an attachment or binding) or a System port. When started, the element achieves the *idle* status.
- `StopElement`. This service stops the execution of an element, *elemID*. The element achieves the *blocked* status when either the tranquillity or quiescence criterion is achieved. In some cases, this

may have the side effect of stopping (i.e. passivating) its adjacent elements.

- `PassivateElement`. This service passivates the execution of an element, *elemID*, with respect to other element, *blockedElement*. This means that *elemID* only processes requests that do not involve interactions with *blockedElement*.

It is important to note that the quiescence of an architectural element instance can only be achieved if it provides a service to stop safely its internal execution. Otherwise, the only option for the Effector aspect is to block all the incoming/outcoming service requests from the instance (i.e. by blocking all the links attached to it), in order to isolate them from the other elements of the architecture. Obviously, this means that pending transactions are interrupted, and then, the instance may achieve an inconsistent state.

For this reason, a requirement for supporting the safe reconfiguration of a System type is that all the architectural elements a System is composed of must be provided with mechanisms for achieving its quiescence. Among the available strategies for implementing the quiescence of running, stateful components (Vandewoude et al., 2007), (Gomaa & Hussein, 2004), (Kramer & Magee, 1990), finally a variation of the tranquillity approach was implemented. The details of this implementation are not described here, but the reader can refer to (Aliaga-Varea, 2008) for further details.

6.4.4.2 Services for Reconfiguration

The Reconfiguration Effector aspect provides four basic services for changing the internal structure of a System instance (see services 4-7 of Figure 6.25):

- `CreateInstance`. This service creates a new instance, *instanceID,* of an architectural type, *typeName*. The initialization parameters required by this type must be provided in *initParams*. The new instance is stopped by default.
- `DestroyInstance`. This service destroys an instance. This is performed by invoking the *destroy* service that PRISMA types provide by default. The instance is removed from memory; if it had connections, they are removed.
- `Connect`. This service establishes a link among two ports of two instances. The two ports must have compatible interfaces; one must provide services, and the other must require services. The service returns the ID of the new link created. As soon as the link is created, the instances can interact by using this link.
- `Disconnect`. This service removes a link, *connectionID*.

The implementation of these services will depend on the underlying infrastructure, and how the code is composed. An extensive study of the different techniques proposed to recompose software can be found in (McKinley et al., 2004a).

In PRISMANET, the implementation of the PRISMA model in .NET, the dynamic recomposition of a System instance is supported due to a combination of different techniques. These techniques are: *wrappers* (which provide the default PRISMA execution model to user-defined architectural elements), *function pointers* (to redirect services among architectural elements) and *aspect weaving* (for the behaviour of architectural elements). An important characteristic is that connections are explicitly separated from architectural elements, which allow us to dynamically change a connection without the need of changing the instance that used such connection. This confirms the importance of considering *connectors* as first-class citizens in software architectures. This separation has been performed by means of *dynamic code generation* (for creating connections), and *publish-subscribe mechanisms* (for sending messages among connections and ports). See (Pérez et al., 2005a), (Costa-Soria, 2005), or (Costa-Soria, 2005a) for further details.

6.4.4.3 Services for Updating, Recovery and Mobility

In addition to the basic services for reconfiguration described above, and the services for achieving quiescence, the Reconfiguration Effector aspect provides other services to support additional features, such as *Dynamic Updating*, *Persistence* & *Recovery*, and *Mobility*.

These features have a common requirement: they need access to the internal state of the instances to manage. The difference is how this state is managed: *dynamic updating* transforms this state to fit the data structures of the new version; *persistence & recovery* stores this state in a permanent location (e.g. in a database or in a file system) for future usage in case a failure occurs; and *mobility* transfers this state over the network in order to recreate the original instance in a different computer. However, these features require that the architectural types being managed (i.e. the type that is going to be dynamically updated or the instances that are going to be stored or moved) provide support to:

1. *State Transfer*: The internal state of an instance should be exportable on demand, for its storage, transmission (over the network) or transformation.
2. *Instance Recreation*: Using a previously exported state, an instance should be recreatable, i.e. re-established again without any noticeable change.

3. *State Mapping*: The state of an instance should be upgradeable to fit the new data structures of a new version of the same type.

These requirements are reflected in our approach as a set of services that must be implemented by each architectural type, in order to allow the dynamic evolution and migration of its instances. Otherwise, if an instance does not implement these services, then obviously its state will be lost when performing a reconfiguration or evolution process.

These services are the following (see the complete signature in Figure 6.26):

- `SerializeState`. This service returns the internal state of an instance. The state is returned in a type-dependent structure encoded as a string (parameter *exportedState*). This state is returned as a string because it makes easy its storage (for recovery) or its transference over different platforms (for mobility).
- `CreateInstanceFromSerializedState`. This service recreates an instance from a previously exported state (parameter *serializedState*). This state must be compatible with the data structures expected by the type that provides this service. The new instance remains in a *Blocked* status until it is activated.
- `ConvertStateFromPreviousVersion`. This service transforms the exported/serialized state of an older type version to the new data structures of the new type version. For these cases where the new type version introduces new data elements, this service allows us providing additional values for these variables that were not present in the old state (parameter *newRequiredValues*).

These services complement each other. To support persistence and mobility of architectural types, only the two first services are needed: for each instance being backed up or moved over the network, its state is serialized and recreated when needed. On the other hand, to support dynamic updating of architectural types we need that: (i) the first service, *SerializeState*, is implemented by the instances of the old type, in order to be able to export its state, and (ii) the new type implements the other two services, to adapt the state of old instances to the new data structures of the new type version and to instantiate the new type with the (transformed) state of old instances.

These services have been grouped in two complementary interfaces: *I_SerializableType* and *I_IncrementalUpdating* (see Figure 6.26). The former is implemented by these architectural types that support instance state migration (i.e. state exportability + instance recreation). The latter is implemented by these architectural types that support state upgrading. In general, all the PRISMA architectural types that are automatically generated implement the

I_SerializableType interface. The *I_IncrementalUpdating* interface is only implemented in PRISMA elements when creating a new, evolved version of another type, in order to support the dynamic updating of its instances, as it will described in Chapter 7.

```
Interface I_SerializableType

// *************************************************************
// Services that a type must provide to support State Migration
// *************************************************************

   SerializeState(output exportedState: string);

   CreateInstanceFromSerializedState(serializedState: string,
      output instanceID: string);

End_Interface I_SerializableType;


Interface I_IncrementalUpdating

// *************************************************************
// Services that a type must provide to support Type Updating
// *************************************************************

   ConvertStateFromPreviousVersion(oldTypeName: string,
      oldState: string, newRequiredValues: list,
      output transformedState: string);

End_Interface I_IncrementalUpdating;
```

Figure 6.26. Required Interfaces for Supporting Dynamic Updating

Since the Effector aspect manages all the elements of a System architecture (i.e. architectural instances), it provides access to the services of *I_SerializableType* and *I_IncrementalUpdating* of each instance (see services 8 to 12 of Figure 6.25). The main difference is that the Effector aspect provides an additional parameter to identify the target instance (*instanceID*) or type (*typeName*) on which invoke the service.

In addition, since we cannot assume that every instance implements these services (e.g. COTS), the Effector aspect also provides additional services for checking whether an instance implements these interfaces or not (see services 8 to 12 of Figure 6.25):

- `IsSerializableType`. This service checks if a type, *typeName*, allows exporting the state of its instances.
- `CanMigrateStateFromOldVersions`. This service checks if a type, *typeName*, provides a function to upgrade the state of previous type versions.

The decisions about how these services are implemented are left to each architectural type. The implementation may differ depending on the different requirements or the target platform.

Next, a few guidelines are presented for guiding this implementation. Each (architectural) type defines how the state of its instances may be exported: what are the elements exported, and how these elements are encoded.

The exported state must include enough information for later restoring the instance: (i) the state, i.e. the values of the attributes that the instance manages; (ii) the instruction pointer, i.e. the next instruction to be executed; and (iii) the connections, i.e. the pointers to the elements that the instance was interacting with. The management of the state of instances has been extensively studied for supporting runtime mobility of software artefacts, and its results can be easily applied for migrating runtime instances to new type versions (i.e. dynamic updating). The reader can refer to the works in this field for more details: either at a technology-dependent level –(Costa-Soria et al., 2006), (Chakravarti et al., 2003)–, or at a high-abstraction level –(Ali, 2008), (Carzaniga et al., 2007), (Mamei & Zambonelli, 2009), (Zachariadis et al., 2006)–.

In PRISMA, the state that is exported from an architectural element is composed of:

1. The attributes of the aspects the architectural element is composed of[31].
2. An ordered list of service requests pending to be processed (i.e. equivalent to the next instruction pointer). These service requests can be either incoming or outcoming.

Note that the information about the clients that have requested these services (i.e. the connections) is not included in the exported state of an architectural element, since this information is implicitly contained in connections (i.e. attachments and bindings), which are managed separately. In addition, recall that the state of an instance can be only consistently exported if the instance is stopped first. Otherwise, the internal state would not be consistent. This has been taken into account: as described before, the stopping of an instance and

[31] Attributes of PRISMA architectural elements are self-contained: they cannot contain references (i.e. memory pointers) to data stored outside the boundaries of an architectural element (i.e. another architectural element). This would make an architectural element dependent of another, which must be avoided. In addition, a data reference implicitly encapsulates a connection; this should be managed explicitly as a connection, in order to manage correctly the architecture dependencies.

its connections is orchestrated by the Reconfiguration Coordination aspect, before the migration of the state of an instance.

The decisions about the structure and encoding of the exported state are left to each (architectural) type. From outside (i.e. the perspective of the Effector aspect), the exported state of an instance is a black box which cannot be inspected or changed. In order to avoid that the internal attributes of this state might be subject to unauthorised inspection or change, the exported state can be encripted differently by each type. This also benefits the independence of architectural types with respect to reconfiguration mechanisms: to support dynamic updating, a type only must provide a function to export the state of an instance and a function to recreate an instance from an exported state.

Then, since the exported state contains all the information of a running instance, it is easy to recreate an instance. The service *CreateInstanceFromSerializedState* is simply a special constructor of the type that, after creating a "normal" instance: (i) initializes its internal attributes with the values defined in the serialized state, (ii) sets the instruction pointer (or in PRISMA, the list of service requests pending to execute), and (iii) leaves the instance in a blocked status, ready for its execution. Then, it is the responsability of the reconfiguration mechanisms (i.e. the Effector aspect) setting correctly the connections and restarting the instance when needed.

Finally, by means of a state transformation process (also known as *state mapping*) the state of old instances is upgraded to fit the data structures of the new type version. This functionality can be provided by an intermediate updating mechanism or by the new type version. We have chosen the new type version as the provider of the state mapping functions because: (i) the developer of the new version knows in advance the data structures of the old version and how these structures have been extended, so it is easier to define the mappings; and (ii) we avoid introducing auxiliary elements in the updating process: only the new type version must be deployed to the target system.

The definition of state mapping functions, such as *ConvertStateFromPreviousVersion*, can be done manually or semi-automatically. Currently, the state mapping functions defined in PRISMA components are defined manually. However, several works have explored the feasibility of automatically generating these state mapping functions. For more details, see section 3.4.2 on page 80.

6.5 The Evolver Component

The previously described aspects provide autonomic reconfiguration capabilities to the composite component (i.e. a System type) they are imported to. That is, only when a System type imports the Monitoring and Reconfiguration Effector aspects, its architecture is made reconfigurable, i.e. it may undergo dynamic changes. The aspects for autonomic reconfiguration have been encapsulated into a component called *Evolver*[32]. The Evolver component represents the ability of a System type to be dynamically reconfigurable. When an Evolver component is imported by a System type, then the PRISMA Model Compiler includes the required dynamic reconfiguration mechanisms in the code of the composite component (which are accessible by means of the Monitoring and Effector aspects).

6.5.1 Structure of the Evolver Component

The Evolver component is a special kind of architectural element. Although it is imported in a System type like another architectural element, it provides services that belong to the *meta-level*. That is, it provides services that introspect and change the architecture within the Evolver component resides. These services can be used by other architectural elements of the System, by simply creating the appropriate connections among them and the Evolver component. This provides us flexibility to decide when a reconfiguration process should start: e.g. when a component invokes a service of the Evolver component, when a certain value is passed through a connection, when an external event is received through a binding, etc.

For instance, Figure 6.27 and Figure 6.28 show the complete architecture of the VisionSystem type, now including an Evolver component, named *VisionSystemEvolver* (depicted in grey in the figure due to its special nature). The VisionSystemEvolver component models (and provides) the reconfiguration infrastructure that is specific for the VisionSystem type. This component has two connections: an attachment to the VisionWatchdog component, and a binding to a port called VisionStatusPort. The attachment allows the VisionSystemEvolver to be directly invoked by the VisionWatchdog component when a failure is detected (see section 6.4.2). The binding allows the VisionSystemEvolver to notify other subsystems of the Agrobot if an unrecoverable failure occurs in the VisionSystem (see also section 6.4.2). This illustrates how an Evolver component can be integrated in the architecture it manages, and interact with other architectural elements when needed.

[32] This name has been chosen because this component also imports other aspects, related to the dynamic evolution of architectural types. See Chapter 7 for further details.

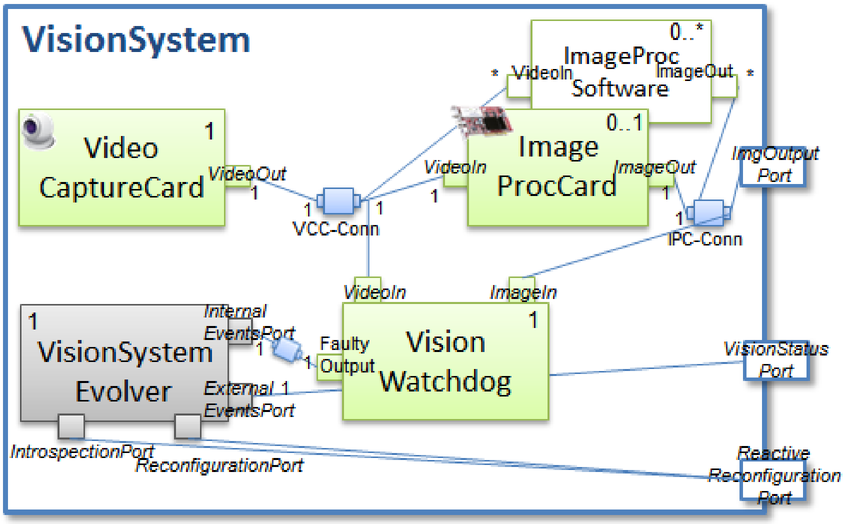


Figure 6.27. Architecture of the VisionSystem type including an Evolver component

```
System VisionSystem

   Import Architectural Elements
      VideoCaptureCard(1,1), ImageProcCard(0,1),
      VCC-Conn(1,1), IPC-Conn(1,1),
      ImageProcSoftware(0,*), VisionWatchdog(1,1),
      VisionSystemEvolver(1,1);

   Ports
      ImgOutputPort : I_ImageProcessingServices;
      VisionStatusPort: I_VisionSystemEvents;
      ReactiveReconfigurationPort: I_VisionSystemReconfigServices;
   End_Ports;

   Attachments
      Att_VCC_VCCConn: VideoCaptureCard.VideoOut(1,1) <-->
            VCC-Conn.VideoIn(1,1);
      Att_VCCConn_IPC:
         VCC-Conn.VideoOut(1,1)<-->ImageProcCard.VideoIn(1,1);
      Att_IPC_IPCConn:
         ImageProcCard.ImageOut(1,1)<-->IPC-Conn.ImageIn(1,1);
      Att_VCCConn_ImgMon:
         VCC-Conn.VideoOut(1,1)<-->VisionWatchdog.VideoOutput(1,1);
      Att_IPCConn_ImgMon:
         IPC-Conn.ImageOut(1,1)<-->VisionWatchdog.ImageOutput(1,1);
      Att_VCCConn_IPCSW:
         VCC-Conn.VideoOut(1,1)<-->ImageProcSoftware.VideoIn(1,*);
      Att_IPCSW_IPCConn:
         ImageProcSoftware.ImageOut(1,*)<-->IPC-Conn.ImageIn(1,1);

      // Example of attachment which enables an internal element to
      // interact with the evolver (e.g.to trigger a reconfig)
      Att_ImgMon_Evolver: VisionWatchdog.FaultyOutputPort(1,1) <-->
         VisionSystemEvolver.InternalEventsPort(1,1);
   End_Attachments;
   Bindings
      Bin_IPCConn: ImgOutputPort(1,1) <--> IPC-Conn.ImageOut(1,1);
      Bin_Evolver: VisionStatusPort(1,1) <-->
                    VisionSystemEvolver.ExternalEventsPort(1,1);
      Bin_Evolver2: ReactiveReconfigurationPort(1,1) <-->
                    VisionSystemEvolver.IntrospectionPort(1,1);
      Bin_Evolver3: ReactiveReconfigurationPort(1,1) <-->
                    VisionSystemEvolver.ReconfigurationPort(1,1);

   End_Bindings;

   /* Constructor definition */
   new ( frameRate: integer, cameraPosition: string,
         timeout: integer)
   {
      ...
   }

   /* Destructor definition */
   destroy()
   {
      ...
   }

End_System VisionSystem;
```

Figure 6.28. PRISMA specification of the VisionSystem Type

The internal structure of an Evolver component is shown in Figure 6.29. The Evolver imports the reconfiguration aspects described before: (i) the Monitoring aspect, which senses the managed architecture; (ii) the Reconfiguration Effector aspect, which acts on the managed architecture; (iii) the Reconfiguration Coordination aspect, which coordinates the execution of reconfigurations; and (iv) the Reconfiguration Analysis aspect, which defines the set of proactive reconfigurations (i.e. autonomously-driven reconfigurations) to perform on the managed architecture. The "managed architecture" is the one where the Evolver component has been integrated (this is depicted as "Architecture" in the figure). The relationships among these aspects are realized by means of weavings, which are depicted in the figure by means of the blue solid circular arrow. The details about these relationships are described in section 6.5.3.

In addition, the Evolver may have three kinds of ports: user-defined ports, reactive reconfiguration ports, and meta-level ports. *User-defined ports* (depicted as green in the figure) are used by the Reconfiguration Analysis aspect to interact with other elements of the system or to be notified about either internal or external events. Reactive reconfiguration ports are the *ReconfigurationPort* and the *IntrospectionPort* (see Figure 6.29). These ports allow us to externally drive a reconfiguration process. Finally, meta-level ports are used by the Reconfiguration Coordination aspect to interact with the meta-level and obtain information about the System type of the architecture being managed (e.g. obtain the cardinality constraints, the valid architectural types and interactions, etc.).

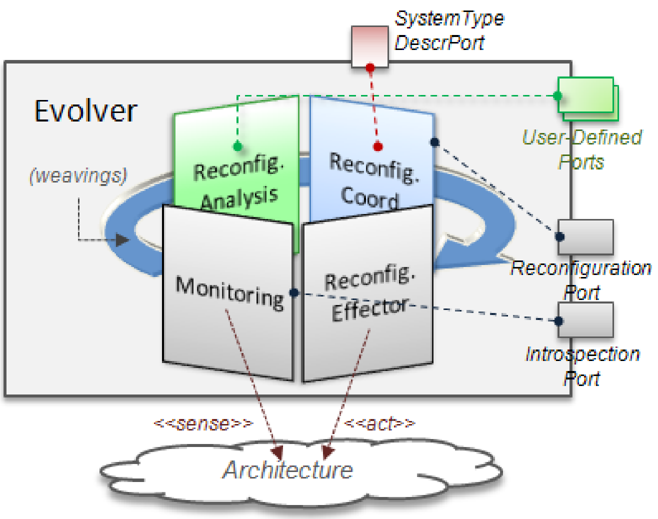


Figure 6.29. Internal Structure of an Evolver component

Figure 6.30 shows a fragment of the PRISMA metamodel and how it has been extended to include the Evolver component and the new kind of aspects (i.e. the reconfiguration aspects). Note the relationship among *Evolver-Component* and *System*: it represents the fact that a PRISMA System type can only have an Evolver component, and that this component manages the System to which it belongs to.

6.5.2 Support for Reactive Reconfigurations

Section 6.4.2 described how proactive reconfiguration (i.e. autonomously-driven reconfigurations) can be supported in our proposal. In this section the support for reactive reconfigurations (i.e. externally-driven reconfigurations) is described.

The support for reactive reconfigurations requires making accessible the introspection and reconfiguration services, so external elements (i.e. another architectural elements or a human) could perform reconfiguration processes. The Evolver component provides support to reactive reconfigurations by default, by means of two ports called *IntrospectionPort* and *ReconfigurationPort*. The former publishes the introspection services provided by the Monitoring aspect (see section 6.4.1.1, Introspection Services). The latter publishes the domain-specific reconfiguration services provided by the Reconfiguration Coordination aspect (see section 6.4.3.1, Domain-Specific Reconfiguration Services). Thus, by using these ports, external elements will be able to perform unanticipated reconfigurations. For instance, by connecting these ports to a component that provides a user interface, the architect could manually perform unanticipated reconfigurations at runtime.

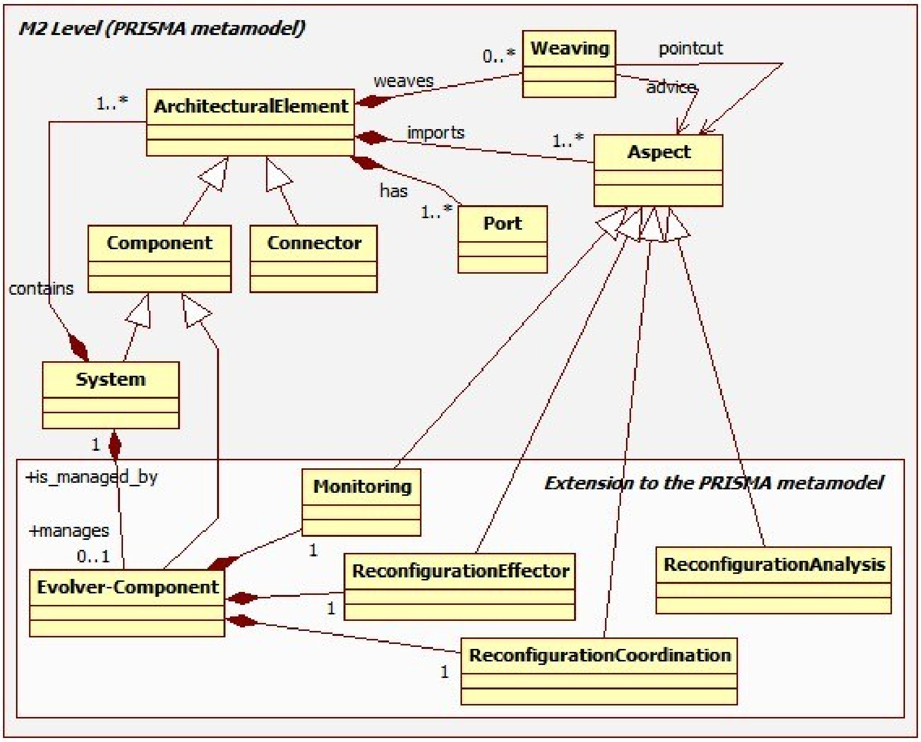


Figure 6.30. Extension of the PRISMA metamodel with the Evolver Component

However, in some situations, the architect may be interested on disabling some services or even do not allow reactive reconfigurations to be performed. In order to provide more control to the architect, both cases are considered. On the one hand, if reactive reconfiguration support is not wanted, then the *IntrospectionPort* and *ReconfigurationPort* should be removed from the default-generated specification of the Evolver component (see Figure 6.36). Thus, external elements could not invoke either introspection or reconfiguration services: reconfigurations will only be performed proactively.

On the other hand, if only a few services are wanted, then the interfaces provided by the *IntrospectionPort* and *ReconfigurationPort* should be modified, removing all the services that are not wanted. In this way, the architect can decide what kinds of introspection/reconfiguration services are allowed or not. Only the services that are included in the interfaces provided by the *IntrospectionPort* and *ReconfigurationPort* will be available to external elements, thus restricting the set of actions which can be performed reactively.

The interfaces provided by the *IntrospectionPort* and *ReconfigurationPort* are automatically generated when an Evolver component is defined. They are named following this pattern: `I_<SysName> IntrospectionServices`, which defines the services provided by the *IntrospectionPort*, and `I_<SysName>ReconfigurationServices`, which defines the services provided by the *ReconfigurationPort*. For instance, the default interfaces that are generated for the VisionSystemEvolver are shown in Figure 6.31 and Figure 6.32. The VisionSystem architect may consider to remove the services *destroy-imageProcCard* and *destroy-imageProcSoftware* to avoid undesired removal of valid instances; or the services related to the connectors to avoid their replacement.

```
Interface I_VisionSystemIntrospectionServices

// ***************************************************************
// Default interface that defines the introspection services
// that will be available for supporting reactive reconfigurations
// on VisionSystem instances.
//
// Remove the services that are not intended to be available
// outside a System instance.
// ***************************************************************

   typeOf(elementID: string, output typeName: string);
   getConfigurationSpecification(output PRISMAConfigSpec: string);

   getAttachedArchElems(archElemID: string, attachType: string,
      output attachedArchElemIDs: list);
   getConnectionsOfArchElem(archElemID: string,output conns: list);
   getConnectionsByType(connectionType: string,output conns: list);
   isAttachment(connID: string, isAtt: boolean);
   isBinding(connID: string, isBind: boolean);

   getArchElementProperties(instanceID: string,
      output properties: list, output portsList: list);
   getPortProperties(archElemID: string, portName: string,
      output isProvided: boolean, output isRequired: boolean,
      output interface: string, output connectionList: list);
   getArchElementInitializationValues(archElemID: string,
      output initValues: list);
   getAttachmentProperties(connectionID: string,
      output instance1: string, output instance2: string);
   getBindingProperties(connectionID: string,
      output sysPort: string, output archElemID: string);

End_Interface I_VisionSystemIntrospectionServices;
```

Figure 6.31. Default Introspection Interface for the VisionSystemEvolver

```
Interface I_VisionSystemReconfigurationServices

// *************************************************************
// Default interface that defines the reconfiguration services
// that will be available for supporting reactive reconfigurations
// on VisionSystem instances.
//
// Remove the services that are not intended to be available
// outside a VisionSystem instance (except those for transaction
// management, which are required to begin/end reconfigurations)
// *************************************************************

// *** TRANSACTION MANAGEMENT –do not remove- ***
   BeginConfigurationTransaction();
   EndConfigurationTransaction();
   RollbackConfigurationTransaction();

//*** ARCHITECTURAL ELEMENTS ***
   //*** videoCaptureCard ***
   getInstances-videoCaptureCard(output instances: list);
   replace-videoCaptureCard(oldInstanceID: string,
            framerate: natural, output newInstanceID: string);

   //*** imageProcCard ***
   getInstances-imageProcCard(output instances: list);
   create-imageProcCard(cameraPosition: string,
            output newInstanceID:string);
   destroy-imageProcCard(instanceID: string);
   replace-imageProcCard(oldInstanceID: string,
            cameraPosition: string, output newInstanceID: string);

   //*** imageProcSoftware ***
   getInstances-imageProcSoftware(output instances: list);
   create-imageProcSoftware(cameraPosition: string,
            output newInstanceID: string);
   destroy-imageProcSoftware(instanceID: string);
   replace-imageProcSoftware(oldInstanceID: string,
            cameraPosition: string, output newInstanceID: string);

   //*** visionWatchdog ***
   getInstances-visionWatchdog(output instances: list);
   replace-visionWatchdog(oldInstanceID: string, timeout: natural,
            output newInstanceID: string);

   //*** VCC-Conn ***
   getInstances-VCC-Conn(output instances: list);
   replace-VCC-Conn(oldInstanceID: string, output newID: string);

   //*** IPC-Conn ***
   getInstances-IPC-Conn(output instances: list);
   replace-IPC-Conn(oldInstanceID: string, output newID: string);

//*** CONNECTIONS: ATTACHMENTS & BINDINGS ***
   //*** Att_VCCConn_IPC ***
   attach-Att_VCCConn_IPC(VCC-ConnID: string,
         ImageProcCardID: string);
   detach-Att_VCCConn_IPC(VCC-ConnID: string,
         ImageProcCardID: string);

   //*** Att_VCCConn_IPCSW ***
```

```
    attach-Att_VCCConn_IPCSW(VCC-ConnID: string,
          ImageProcSoftwareID: string);
    detach-Att_VCCConn_IPCSW(VCC-ConnID: string,
          ImageProcSoftwareID: string);

    //*** Att_IPC_IPCConn ***
    attach-Att_IPC_IPCConn(ImageProcCardID: string,
          IPC-ConnID: string);
    detach-Att_IPC_IPCConn(ImageProcCardID: string,
          IPC-ConnID: string);

    //*** Att_VCCConn_IPC ***
    attach-Att_IPCSW_IPCConn(ImageProcSoftwareID: string,
          IPC-ConnID: string);
    detach-Att_IPCSW_IPCConn(ImageProcSoftwareID: string,
          IPC-ConnID: string);

End_Interface I_VisionSystemReconfigurationServices;
```

Figure 6.32. Default Reconfiguration Interface for the VisionSystemEvolver

6.5.3 Weaving the Reconfiguration Aspects Together

As described in the previous sections, the Reconfiguration Aspects use (or require) the execution of services from other aspects. The Reconfiguration Analysis aspect uses services provided by the Monitoring aspect and the Reconfiguration Coordination aspect to define configuration triggers and configuration transactions. The Reconfiguration Coordination aspect uses introspection services and low-level reconfiguration services to drive the safe execution of configuration transactions.

However, a characteristic of PRISMA aspects is that they cannot have direct invocations of services from other aspects: specifically, calls like "`aspectName.serviceName(parameters)`" are avoided in PRISMA. This avoids an aspect specification to be dependent of other aspects. The invocations of services among aspects can be addressed in PRISMA by means of *out services* and weavings. *Out services* are used to declare the services that are required from other elements. Then, these services can be hooked either to ports, if the services are provided by another architectural element, or to aspects (through weavings), if the services are provided by another aspect. In this way, an aspect only declares the set of services that are required, but not which element provides these services and how they are provided. This benefits the independence of aspects and its maintenance.

Therefore, to define a service invocation among aspects: (i) the service must be declared in the *Service* section of the client aspect as an *out service*, and (ii) a weaving must be defined among this service and the service provided by the target aspect. An advantage of using weavings to hook services among aspects is that they do not need to have the same name and signature, since functions can be used to adapt incompatible signatures (Guillén-Martín, 2007, pp. 43-60).

For instance, each configuration transaction defined in a Reconfiguration Analysis aspect uses several domain-specific reconfiguration actions, which are provided by the Reconfiguration Coordination aspect. For this reason: (i) the Reconfiguration Analysis aspect declares all the domain-specific reconfiguration actions used as *out services* (see Figure 6.33), and the Evolver component defines the weavings among these services and the services provided by the Reconfiguration Coordination aspect (see Figure 6.34). This is done similarly for the synchronization among the Reconfiguration Coordination aspect and the Monitoring and Reconfiguration Effector aspects.

```
ReconfigurationAnalysis Aspect VisionSystemAnalysisServices
...
   Services
   // *** Introspection services *********************************
   out typeOf(instanceID: string, output typeName: string);
   out getArchElementProperties(instanceID: string,
         output properties: list, output portsList: list);
   out getPortProperties(archElemID: string, portName: string,
         output isProvided: boolean, output isRequired: boolean,
         output interface: string, output connectionList: list);
   out getAttachedArchElems(archElemID: string,attachType:string,
         output attachedArchElemIDs: list);
   // ... [more introspection services]

   // *** Domain-specific reconfiguration actions *****************
   out create-ImageProcCard(cameraPosition: string,
            output newInstanceID : string);
   out create-ImageProcSoftware(cameraPosition: string,
            output newInstanceID : string);
   out destroy-imageProcCard(instanceID : string);
   out destroy-imageProcSoftware(instanceID : string);
   // ... [more introspection services]

   // *** Services for Event Interception ************************
   out beforeEvent(eventName: string, output eventParams: list);
   out afterEvent(eventName:string, output eventParams:list);
   out insteadOfEvent(eventName: string, condition: string,
         replacingService: string, output eventParams:list);

   // *** Services for Selective Element Starting/Stopping ********
   out StartElement(instance-ID: string);
   out StopElement(instance-ID: string);
...

End ReconfigurationAnalysis Aspect
         VisionSystemAnalysisServices;
```

Figure 6.33. Example of declaration of out services of a Reconf. Analysis aspect

```
Component VisionSystemEvolver
...
   Weavings
   //*** Weavings: RECONFIGURATION_ANALYSIS --> MONITORING *******
1  Monitoring.beforeServiceRequest(*, eventName, eventParams)
      insteadOf
   VisionSystemReconfigurationAnalysis.beforeEvent(eventName,eventParams);

2  Monitoring.insteadOfServiceRequest(*, eventName, condition,
            replacingService, eventParams)
      insteadOf
   VisionSystemReconfigurationAnalysis.insteadOfEvent(eventName, condition,
            replacingService, eventParams);

3  Monitoring.afterServiceRequest(*, eventName, eventParams)
      insteadOf
   VisionSystemReconfigurationAnalysis.afterEvent(eventName, eventParams);

4  Monitoring.getAttachedArchElems(archElemID, attachType,
            attachedArchElemIDs)
      insteadOf
   VisionSystemReconfigurationAnalysis.getAttachedArchElems(archElemID,
            attachType, attachedArchElemIDs);
   //... [more introspection services]

5  Monitoring.getArchElementInstances("self", new list[sysID] )
      insteadOf
   VisionSystemReconfigurationAnalysis.getSystemInstanceID(sysID);

6  Monitoring.getArchElementInstances("VideoCaptureCard", list-IDs)
      insteadOf
   VisionSystemReconfigurationAnalysis.
            getInstances-videoCaptureCard(list-IDs);

   //... [same weavings for each architectural type]

   //*** RECONFIGURATION_ANALYSIS --> RECONFIGURATION EFFECTOR ***
7  ReconfigurationEffector.StartElement(instance-ID)
      insteadOf
   VisionSystemReconfigurationAnalysis.StartElement(instance-ID);

8  ReconfigurationEffector.StopElement(instance-ID)
      insteadOf
   VisionSystemReconfigurationAnalysis.StopElement(instance-ID);

   //*** RECONFIGURATION_ANALYSIS --> RECONFIG. COORDINATION *****
9  VisionSystemReconfigurationServices.BeginConfigurationTransaction()
      insteadOf
   VisionSystemReconfigurationAnalysis.TRANSACTION.BEGIN;

10 VisionSystemReconfigurationServices.EndConfigurationTransaction()
      insteadOf
   VisionSystemReconfigurationAnalysis.TRANSACTION.END;

11 VisionSystemReconfigurationServices.RollBackConfigurationTransaction()
      insteadOf
   VisionSystemReconfigurationAnalysis.TRANSACTION.ROLLBACK;

12 VisionSystemReconfigurationServices.create-ImageProcCard(params,
            newInstanceID)
      insteadOf
   VisionSystemReconfigurationAnalysis.
            create-ImageProcCard(params,newInstanceID);

13 VisionSystemReconfigurationServices.destroy-imageProcCard(instanceID)
      insteadOf
   VisionSystemReconfigurationAnalysis.destroy-imageProcCard(instanceID);

   // ... [similar weavings among the VisionSystemReconfigurationAnalysis
   // aspect and the VisionSystemReconfigurationServices aspect]
   End_Weavings;
...
End_Component VisionSystemEvolver;
```

Figure 6.34. Example of weaving definitions in the Evolver component

Note in the definition of weavings that some source services do not share the same signature of the target service: they have different name or even different parameters. For instance, the weaving n°1 (Figure 6.34, n°1) binds two services with different name and number of parameters. The target service *beforeServiceRequest* has one of its parameters set to the default value "*", which means that the ServiceRequest can be intercepted from any element of the architecture (see section 6.4.1.3-Event Interception Services). The weaving n°5 is similar: the source and target services have different name and kind of parameters. The source service *getSystemInstanceID* requires the ID of the System instance which is being managed. The target service *getArchElementInstances* returns the IDs of a certain type of architectural element. In this case, the "self" keyword identifies the type of the System type, and then the service returns the ID of the System instance which is provided to the source service by means of parameter matching. Another example is the weaving n°9, which *weaves* the beginning of a configuration transaction (identified with the special keyword `TRANSACTION.BEGIN`) to the service BeginConfigurationTransaction provided by the Reconfiguration Coordination aspect. Weavings 10 and 11 behave in a similar way, to end or rollback a configuration transaction, respectively.

Since the Reconfiguration Analysis aspect is defined by the architect, he should declare the necessary out services and weavings corresponding to the introspection and reconfiguration services used in the specification. In order to alleviate the architect from these tasks, the synchronizations among reconfiguration aspects (i.e. the declaration of out services + the definition of weavings) are automatically generated by default when a new Reconfiguration Analysis aspect is instantiated.

6.5.4 Evolver Specification

The structure of an Evolver type and its relationships with the PRISMA metamodel is depicted in Figure 6.35. This figure reflects the different kind of artefacts that an Evolver type is made of, which are depicted in different colors for claritiy purposes (see Figure 6.35):

- *Generic* (or base) *elements*: elements that define the common reconfiguration behaviour for all the Evolvers. They are depicted in yellow.
- *Automatically-generated elements*: elements that define the reconfiguration behaviour that is specific for a concrete System type. They are depicted in green: *<SysName>EvolverMechanisms*, *<SysName>ReconfigurationServices*, *<SysName>AnalysisServices*, *IntrospectionPort*, and *ReconfigurationPort*.
- *Customizable generated elements*: elements that can be modified or customized to fit the architect's reconfiguration needs. They are depicted in blue: *<SysName>Evolver*, *<SysName>Reconfiguration AnalyisisAspect*, *I_<SysName>IntrospectionServices*, and *I_<SysName>ReconfigurationServices*

This decomposition is due to the different rates of change exhibited by these elements: their separation in different artefacts improves its maintenance and further evolution (Mens & Wermelinger, 2002). Note that almost all the automatically-generated elements (except ports) are prefixed with the name of the System type that is managed, <*SysName*>. This is because they are specifically generated for a specific System type (i.e. *SysName*) and are not valid for a different type.

These elements are described in the next sections.

6.5.4.1 Evolver Template: The User-Defined Part

The template that is used to generate the specification of a specialized (i.e. System-specific) Evolver component type is shown in Figure 6.36. The name of an Evolver component type is prefixed with the name of the System type that it manages, *SysName*. This is because it contains automatically-generated structures which are specific for a System type, and are not valid for a different System type.

The specification of an Evolver type provides the following elements:

- The ports *IntrospectionPort* and *ReconfigurationPort*, which provide reactive reconfiguration support. These ports are bound to a Monitoring aspect, called *MonitoringAspect*, and to a Reconfiguration

Coordination aspect, called *<SysName>ReconfigurationServices*. These ports can be removed from the Evolver specification to forbid the use of (external) reactive reconfigurations.

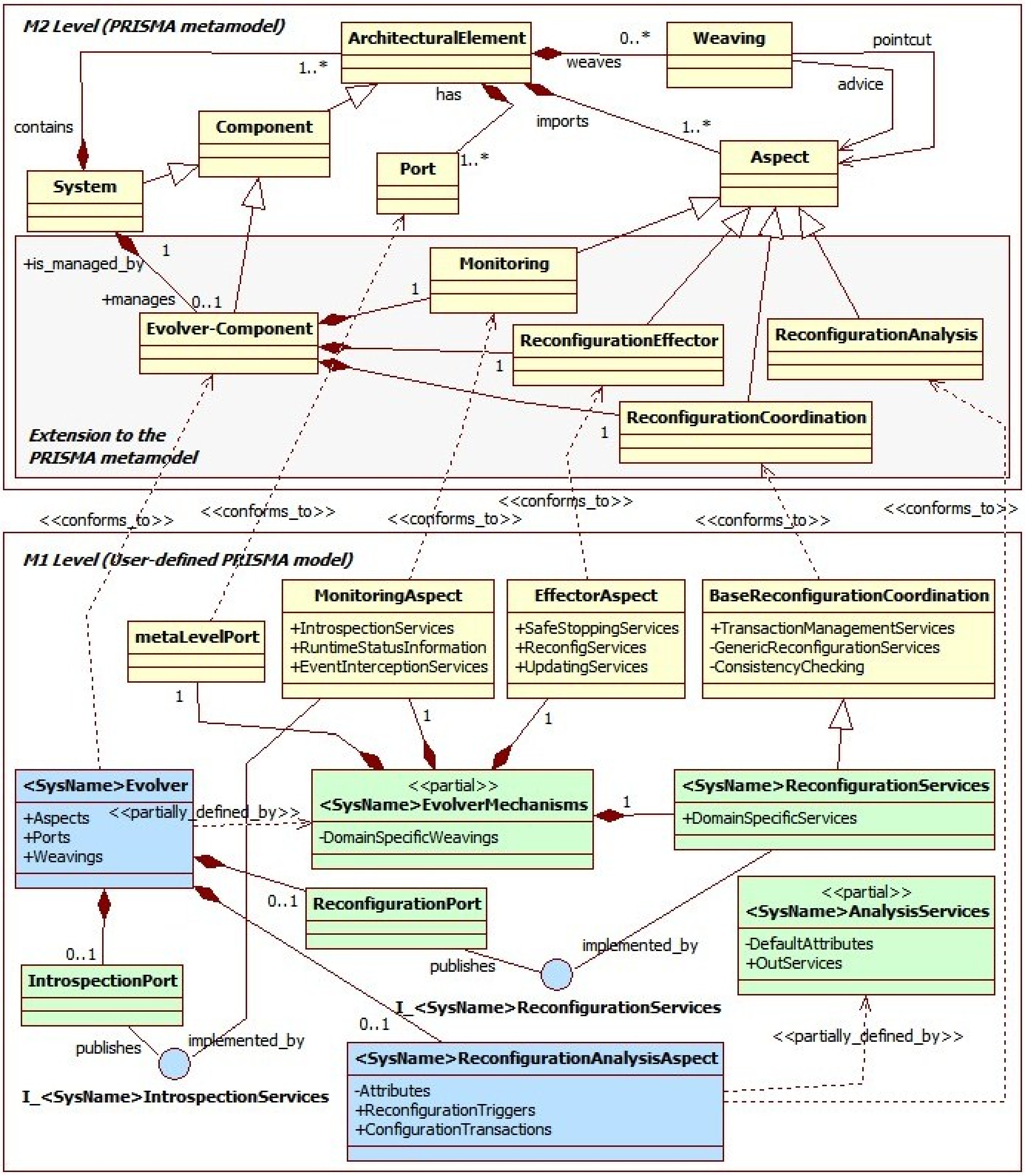


Figure 6.35. Metamodel of the Evolver Component

- The interfaces *I_<SysName>IntrospectionServices* and *I_<SysName>ReconfigurationServices*, which define the services that are published through the *IntrospectionPort* and *ReconfigurationPort*. These interfaces can be customized by the architect, as described in section 6.5.2, by removing unwanted services.

- An empty ReconfigurationAnalysis aspect, called `<SysName>ReconfigurationAnalysisAspect`, which must be completed by the architect to include proactive reconfiguration support. This aspect is partially defined by another artefact, called *<SysName>AnalysisServices*, which defines the *out services* that are required to weave the Analysis aspect with the Reconfiguration Coordination aspect, as described in section 6.5.3.

```
Evolver-Component <SysName>Evolver
   is partially defined by <SysName>EvolverMechanisms

   // Proactive reconfiguration support
   ReconfigurationAnalysis Aspect
         import <SysName>ReconfigurationAnalysisAspect;

   // Additional aspects can be imported if needed. Import below.

   Ports
      // Ports for providing Reactive Reconfiguration Support
      // (Remove them if reactive reconfig. is not needed)
      ReconfigurationPort :
         I_<SysName>ReconfigurationServices,
         Played_Role <SysName>ReconfigurationServices.RECONFPLANS;
      IntrospectionPort : I_<SysName>IntrospectionServices,
         Played_Role MonitoringAspect.INTROSPECT;

      // Define here additional ports
   End_Ports;

   Weavings
      // Define here the weavings among user-defined aspects
   End_Weavings;

   // Initialization and Destruction services
   new() {
      // User-defined initialization here
   }

   destroy() {
      // User-defined destruction here
   }

End_Evolver-Component <SysName>Evolver;
```

Figure 6.36. Template for Evolver component types

The Evolver specification can be customized to fit the architect's reconfiguration needs, by adding additional aspects, ports and weavings. For instance, an example of a customized Evolver specification is the

VisionSystemEvolver, the Evolver of the VisionSystem type, which is provided in the appendix A.2.2.1.

6.5.4.2 Evolver Mechanisms: The Generated Functionality

Note that the Evolver template shown in Figure 6.36 does not include any reference to the specific reconfiguration mechanisms of the Evolver, such as the Monitoring aspect, the Reconfiguration Effector aspect, the Reconfiguration Coordination aspect or the weavings among them. These elements have been explicitly separated from the Evolver specification to isolate user-defined specifications (i.e. the customizations made to the Evolver type by the architect) from automatically-generated specifications (i.e. the mechanisms that provide reconfiguration support). This is to avoid introducing inconsistencies to each other.

On the one hand, the user (i.e. the architect) should not modify automatically-generated specifications: they are regenerated each time the System type is modified, so any change performed on them will be lost. In addition, generated specifications define the synchronizations among reconfiguration aspects; if they are modified, reconfiguration inconsistencies may be introduced. For instance, if configuration actions are modified, or additional ones are manually added, the System type constraints may be violated. On the other hand, user-defined specifications must be defined separately, to avoid that the compiler could remove them when regenerating the specification again, thus losing user-defined functionality.

In order to combine user-defined specifications with automatically-generated specifications but also keeping them separated for code-generation purposes, we have used the concept of partially defined artefacts:

> *A partially-defined artefact is a software artefact (i.e. a type, a class, a specification) that is splitted in several parts, which are combined in the compiling process.*

This notion is particularly useful to keep separated user-defined specifications from automatically-generated specifications, whereas both define the same artefact.

Therefore, the Evolver mechanisms that are automatically-generated and cannot be customized by the architect have been encapsulated in a partial specification called *<SysName>EvolverMechanisms*. This partial specification complements the customizable *<SysName>Evolver* specification: see the header "is partially defined by" in the template of Figure 6.36. A question that the reader may be thinking about is why inheritance has not been used to separate generated code from user-defined code. The reason is that generated code uses artefacts that are declared by the user; for instance, the Reconfiguration Analysis aspect, which the generated code weaves to the other reconfiguration aspects. And, since the Reconfiguration Analysis aspect is an optional element, it cannot be declared by the generated code.

<SysName>EvolverMechanisms is an automatically-generated partial specification that assembles the reconfiguration elements that characterize an Evolver type. These elements are the following:

- The aspects that monitorize and change the managed architecture: *MonitoringAspect* and *EffectorAspect*.
- The port that is required to introspect the managed System type, *metaLevelPort*.
- A specialized (i.e. domain-specific) Reconfiguration Coordination aspect, called *<SysName>ReconfigurationServices*. This aspect provides the set of available configuration actions for changing *<SysName>* and inherits the generic reconfiguration behaviour (i.e. generic reconfiguration services, transactional management and consistency checking) from a predefined aspect called *BaseReconfigurationCoordination*.
- The necessary relationships to weave these elements together (i.e. the weavings among aspects), *DomainSpecificWeavings*.

For illustrative purposes, Figure 6.37 shows a fragment of the generated EvolverMechanisms for the VisionSystem type. The complete specification is provided in appendix A.2.2.1.

```
Component VisionSystemEvolverMechanisms
   is partial

// ************************************************************
// Automatically-generated partial Evolver specification for
// VisionSystem instances.
//
// This part contains the weavings among the different
reconfiguration
// mechanisms. It is regenerated each time the VisionSystem type
changes.
// CANNOT BE CHANGED BY THE USER
// ************************************************************

   ReconfigurationCoordination Aspect
         import VisionSystemReconfigurationServices;
   Monitoring Aspect import MonitoringAspect;
   ReconfigurationEffector Aspect import EffectorAspect;

   Ports
```

```
      SystemTypeDescrPort : I_CompositeTypeDescription,
         Played_Role  VisionSystemReconfigurationServices.
                      META-TYPEDESCRIPTION;
   End_Ports;

   Weavings

   //*** Weavings: RECONFIGURATION_ANALYSIS --> MONITORING ***
   Monitoring.beforeServiceRequest(*, eventName, eventParams)
      insteadOf
   VisionSystemReconfigurationAnalysis.beforeEvent(eventName,
            eventParams);

   Monitoring.insteadOfServiceRequest(*, eventName, condition,
            replacingService, eventParams)
      insteadOf
   VisionSystemReconfigurationAnalysis.insteadOfEvent(eventName,
            condition, replacingService, eventParams);

   //... More introspection services


   //*** RECONFIGURATION_ANALYSIS --> RECONFIGURATION EFFECTOR ***
   ReconfigurationEffector.StartElement(instance-ID)
      insteadOf
   VisionSystemReconfigurationAnalysis.StartElement(instance-ID);

   ReconfigurationEffector.StopElement(instance-ID)
      insteadOf
   VisionSystemReconfigurationAnalysis.StopElement(instance-ID);

   //*** RECONFIGURATION_ANALYSIS --> RECONFIG. COORDINATION *****
   VisionSystemReconfigurationServices.BeginConfigurationTransaction()
      insteadOf
   VisionSystemReconfigurationAnalysis.TRANSACTION.BEGIN;

   ...
```

Figure 6.37. Fragment of VisionSystemEvolverMechanisms

6.5.4.3 Consistence of Generated Code

These reconfiguration facilities are automatically generated each time the System type is modified. If an architectural element or connection is introduced in the System type, the corresponding configuration actions (i.e. create-, destroy-, attach-, …) are automatically added to the specialized Reconfiguration Coordination aspect (i.e. the aspect *<SysName>ReconfigurationServices*), so the new architectural element or connection could be instantiated. Similarly, if an architectural element or connection is removed, the corresponding configuration actions are automatically removed from the specialized Reconfiguration Coordination aspect, avoiding the instantiation or connection of removed elements.

Obviously, these modifications may leave user-defined reconfiguration specifications inconsistent. Specifically, inconsistencies may emerge when a user-defined configuration transaction performs operations that, due to modifications in the System type, are invalid. There are two kinds of invalid operations: (i) the use of configuration actions that are no further available, or (ii) the violation of cardinality constraints. The former may emerge when an architectural type or connection type has been removed from the System type, but reconfiguration specifications still use reconfiguration services related to the removed elements (i.e. create-, destroy-, attach-, detach-). The latter may emerge when the cardinality of architectural types or connections is changed. Depending on the kind of inconsistency introduced, it will be managed at compile-time or at runtime.

The first kind of inconsistency, the use of invalid configuration actions, is automatically controlled by the PRISMA model compiler. Since configuration actions are defined as services, in case one is no more available (e.g. we cannot create instances of an element or we cannot attach two elements), then all the user-defined reconfiguration specifications that invoke such disabled actions would not compile. The user will be aware that its specifications are not valid and should be modified. This is another advantage of using domain-specific reconfiguration actions for defining reconfiguration policies.

The second kind of inconsistency, the violation of cardinality constraints, is managed at runtime by the Reconfiguration Coordination aspect. If a user-defined configuration transaction violates the cardinalities defined by the System type, then the preconditions of the Reconfiguration Coordination aspect simply forbid the execution of these transactions, thus avoiding invalid reconfigurations.

Thus, by means of code-generation techniques, the set of domain-specific reconfiguration services provided by an Evolver type always reflect the allowed operations that can be performed on a System type. In case a System type changes, the set of allowed reconfiguration operations (i.e. configuration actions) is updated accordingly. This operation prevents the execution of invalid user-defined reconfiguration policies: reconfiguration policies that use reconfiguration operations that are no further available due to the changes performed to the System type.

6.6 Example: autonomic reconfiguration in the VisionSystem architecture

Finally, to provide a summary of the entire autonomic reconfiguration process, this section describes the execution of a reconfiguration scenario in a VisionSystem composite instance.

Figure 6.38 shows the configuration of one of the VisionSystem composite instances: the RightCamera instance.

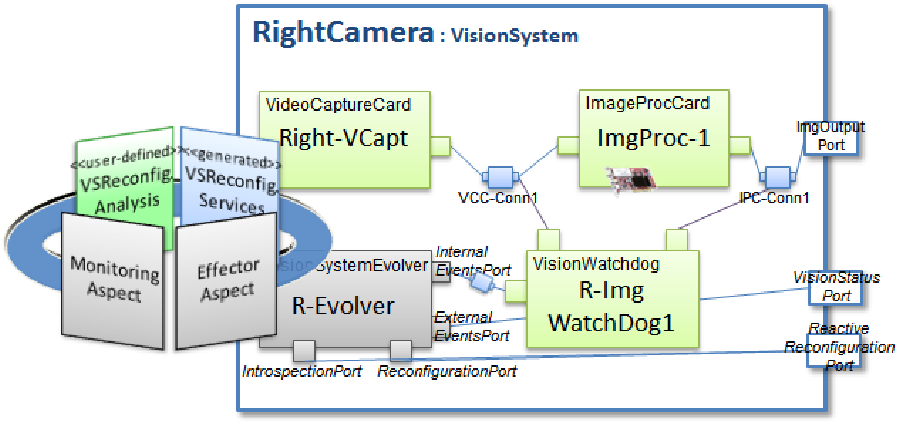


Figure 6.38. Architecture of the RightCamera composite instance, with the detail of the reconfiguration aspects provided by an Evolver instance

This configuration instantiates the different components defined in the pattern of the VisionSystem type (see Figure 6.27, page 223): *Right-VCapt*, an instance of the component *VideoCaptureCard*; *ImgProc-1*, an instance of the component *ImageProcCard*; *R-ImgWatchDog1*, an instance of the component *VisionWatchdog*; *R-Evolver*, an instance of the component *VisionSystemEvolver*; and their respective connectors *VCC-Conn1* and *IPC-Conn1*. The complete specification of this configuration in PRISMA ADL is provided in appendix A.1.4.2 (see page 385).

A reconfiguration scenario describes how the architecture of a composite instance is reconfigured at runtime when a certain situation or event takes place. This reconfiguration scenario can be executed *proactively*, if it was defined at design-time, or *reactively*, if it is defined and introduced at runtime. The difference is that proactive reconfiguration scenarios are specified inside a *ReconfigurationAnalysis* aspect, whereas reactive reconfiguration scenarios are introduced by using the services provided by the ports *IntrospectionPort* and *ReconfigurationPort* of an Evolver instance (for more details, see section 6.5.2).

In both cases, reconfigurations are internally executed by invoking the reconfiguration services provided by a Reconfiguration Coordination aspect.

Next, to illustrate how a reconfiguration process is internally executed inside an Evolver component, one of the reconfiguration scenarios described in section 5.5.1 (see page 150) is used: the dynamic reconfiguration after the failure of an *ImageProcCard* instance. The supervision of *ImageProcCard* instances (among others) is performed by the component *VisionWatchdog*. In case misbehaviour is detected, the *VisionWatchdog* component sends a *faultyOutput* event to notify others about the failure. This event is intercepted by the Evolver component, which evaluates, by means of the *VSReconfigurationAnalysis* aspect, whether a reconfiguration must be performed in response to the event or not.

One of the reconfiguration scenarios defined in this aspect, *RepairImageProcessingUnit*, describes that in case of the failure of an *ImageProcCard* instance it must be replaced by an instance of the component *ImageProcSoftware*, which provides an alternative implementation of the same behaviour. The specification of this reconfiguration in PRISMA ADL has been presented in Figure 6.11 (see page 186), so it is not recreated again. The execution of this scenario, and the sequence of interactions among the reconfiguration aspects, is shown in Figure 6.39.

The first interaction is performed by the *VSReconfigurationAnalysis* aspect: when it is instantiated, it defines the event that will trigger a reconfiguration process: “faultyOutput”. This is done through the service beforeEvent! (see step 1 in Figure 6.39), which is actually implemented by the *MonitoringAspect*. This aspect monitorizes the services that take place in the architecture of a *VisionSystem* instance; if it intercepts the desired event, then it activates the trigger defined in the *VSReconfigurationAnalysis* aspect (see sequence of interactions from 2 to 5). When the *VSReconfigurationAnalysis* is activated (i.e. it has been notified about an event it was subscribed for), then the reconfiguration transaction *repairImageProcessingUnit* is initiated (see step 6). This implicitly activates the execution of the service *beginConfigurationTransaction*, provided by the *VSReconfigurationServices* aspect and that prepares the architecture for being reconfigured at runtime (see step 7).

Next, for each reconfiguration operation defined in the reconfiguration transaction, the *VSReconfigurationServices* performs the corresponding actions (see section 6.4.3.3 for more details).

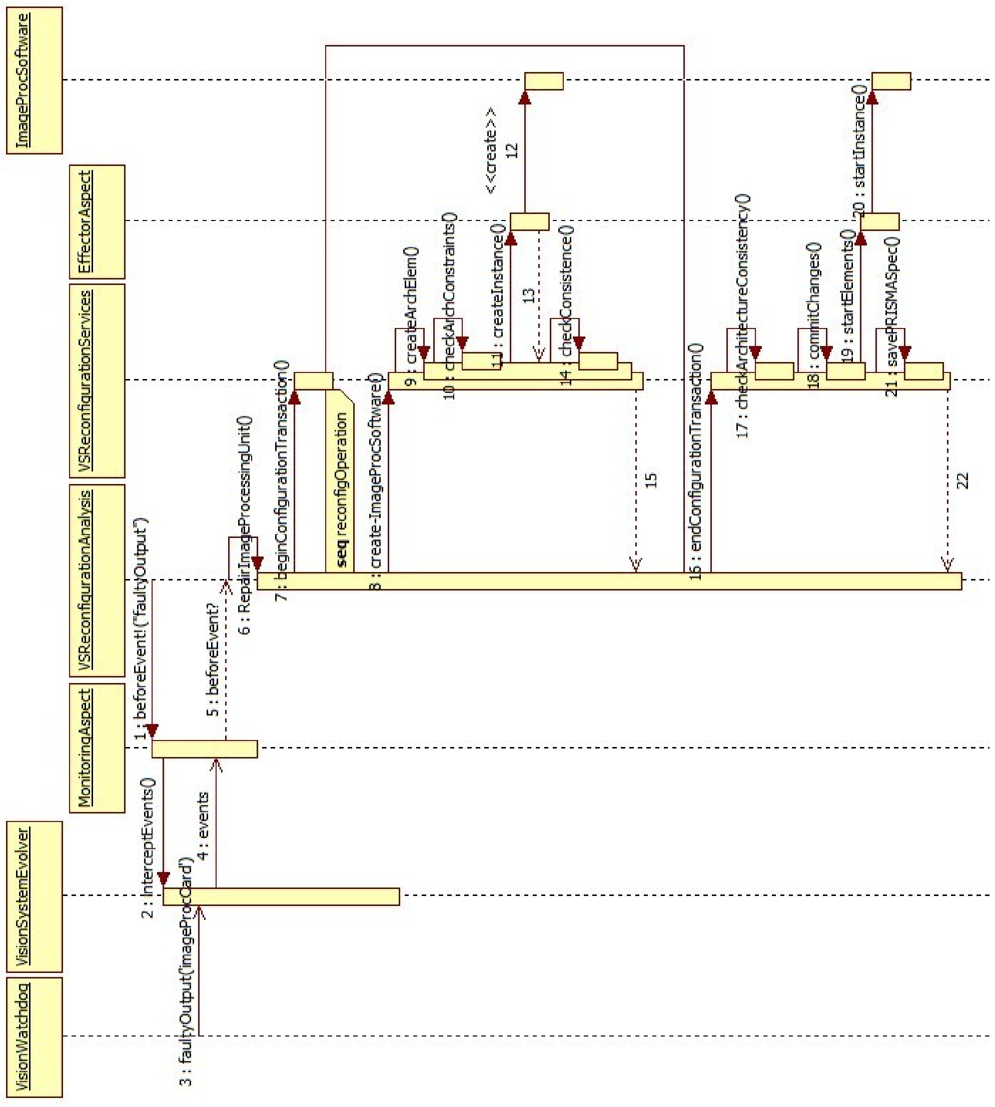


Figure 6.39. Sequence of interactions among reconfiguration aspects as a result of a configuration transaction

The figure shows the sequence of actions corresponding to the instantiation of a new architectural type: when the domain-specific reconfiguration action *create-ImageProcSoftware* is executed by the *VSReconfigurationAnalysis* aspect (see step 8), then the *VSReconfigurationServices* aspect initiates the generic reconfiguration service *createArchitecturalElement* (see steps 9 to 14). This service checks that the architectural constraints are fulfilled (i.e the type is valid, and the maximum and minimum cardinality is not violated), then creates the instance (which is performed by the *EffectorAspect*, see steps 11-13) and checks that the operation has been successfully performed (service *checkConsistence*). This is similar to the other reconfiguration operations, except for removals, where a safe stopping operation is performed first.

Finally, after the successful execution of all the reconfiguration operations, the reconfiguration transaction is committed. This is implicitly initiated when the last reconfiguration operation is finished, and it is performed by the *VSReconfigurationServices* aspect (see steps 15 to 22). This service first checks the consistency of the architecture (see step 17), and if it is valid, then it commits the changes (i.e. deletions and replacements are confirmed) and starts the execution of all the stopped elements (see step 19 and 20). Finally, it generates the PRISMA ADL specification for making changes permanent (see step 21).

Note that in Figure 6.39, the interactions among aspects are depicted as direct calls. However, this has been depicted in this way for simplicity reasons. The reader must take into account that interactions among aspects are not performed through direct calls, but through weavings: a weaving intercepts an aspect service (e.g. *create-ImageProcSoftware*) and executes the code from another aspect instead. In this way, aspects do not have direct references among them, thus improving reuse and maintenance.

As a result, the architecture of the *RightCamera* is reconfigured at runtime: the failing *ImageProcCard* instance is removed and replaced by an instance of a different component, an instance of the component *ImageProcSoftware*. The resulting configuration is shown in Figure 6.40.

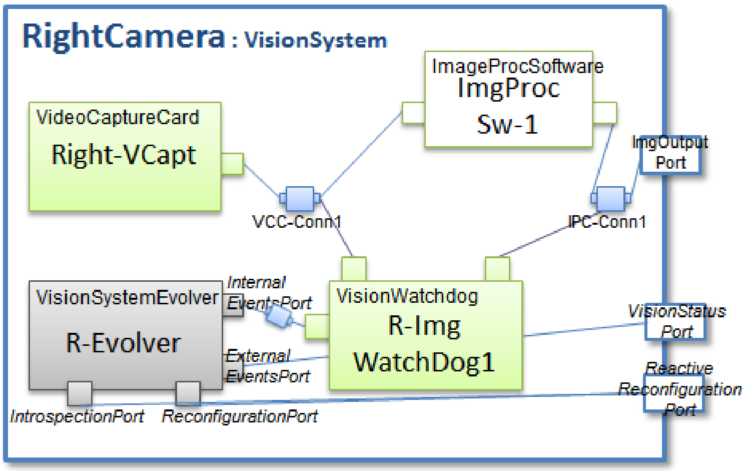


Figure 6.40. Architecture of the RightCamera composite instance, after the execution of the RepairImageProcessingUnit reconfiguration process

This is performed transparently to the architect, without the need of dealing with low-level details, such as consistency management issues.

6.7 Conclusions & further works

This work provides four contributions to the design of autonomous dynamically reconfigurable systems. First, it defines a model to bridge the gap among high-level reconfiguration specifications and low-level supporting mechanisms. Second, it provides each composite component with self-reconfiguration capabilities to autonomously change its internal composition. Third, it considers the support for both reactive and proactive reconfigurations, to achieve a better level of flexibility. Fourth, it explicitly separates reconfiguration concerns from other concerns of the system, to improve their maintainability and reuse. And fifth, it provides composite instances with reconfiguration plasticity to tolerate changes without breaking the design decisions defined in their types.

6.7.1 Conclusions

This chapter has described one perspective of our work: the support for autonomic reconfiguration of hierarchical software architectures. Each subsystem (i.e. a System) is provided with dynamic reconfiguration mechanisms to proactively change its internal structure according to either internal or external stimuli. These mechanisms are provided by four aspects:

- The different stimuli are captured by the Monitoring aspect, which monitorizes the architecture and intercepts events. These events can be internal, if they are triggered by the elements of the System instance being managed, or external, if they come from outside the context of the System instance (i.e. they are delivered through System ports)
- The selection of stimuli (i.e. reconfiguration triggers) and the reactions to these stimuli (i.e. reconfiguration operations) are defined by a different aspect, the Reconfiguration Analysis aspect;
- The orchestration of the reconfiguration process and its consistence is performed by the Reconfiguration Coordination aspect; and
- The changes to the architecture are performed by the Reconfiguration Effector aspect, the mediator with the execution platform.

As a result, the separation of concerns is enforced by isolating: user-defined reconfiguration policies (Reconfiguration Analysis aspect), domain-specific reconfiguration facilities (Reconfiguration Coordination aspect) and reconfiguration mechanisms (Monitoring and Reconfiguration Effector aspects). The user only deals with high-level reconfiguration actions (e.g. *create-*, *attach-*, *detach-*, ...), without taking care of low-level reconfiguration actions (e.g. quiescing elements, state migration, selective stopping, etc.). This results in high-abstraction level, easier to understand, and easy maintainable reconfiguration specifications.

Another advantage of our approach is that the information about the running system is obtained at runtime, by means of the introspection services provided by the Monitoring aspect. Thus, there is no need for keeping a runtime model of the system, since system information is obtained when needed.

In addition, the interception services provided by the Monitoring aspect are very powerful. They do not only notify about services that are requested in the architecture, but also about the execution of reconfiguration services. The interception of reconfiguration events is a very useful feature to keep visual representations or models of a system updated as soon as the system changes. If an architectural instance is created, removed, or its connections changed, then the visual model should be notified to represent this situation. This is better than periodically obtaining the complete specification of the running system (by means of the *getConfigurationSpecification* service) and checking changes.

For each System type, its reconfiguration code is centralized in a special component called Evolver. This avoids the scattering of this code among the architecture, as in decentralized approaches. Thus, a reconfigurable System will have a fixed part, i.e. the Evolver component, and a variable part where the Evolver will act upon, i.e. all the other architectural elements and connections of the System. The Evolver encapsulates the different aspects and integrates the domain-specific behaviour (i.e. reconfiguration policies and domain-specific reconfiguration operations) with the domain-independent behaviour (event interception, architecture introspection, and architecture modification services). This is made by means of automatically-generated code, which defines the necessary synchronization code (weavings and out services) among aspects.

Thus, this approach provides a software architecture with the following properties: (i) flexibility, due to the use of both proactive and reactive dynamic reconfigurations; (ii) maintainability, because aspect-oriented techniques are used to separate reconfiguration concerns from other concerns; and (iii) scalability, because management is decentralized to each composite component.

Another contribution of our approach is that the reconfigurations performed on a System instance are limited by the constraints defined in its System type. This is to avoid that System instances, due to several reconfigurations, could lose the conformance with their System type, and thus their integration with other systems. A System type defines which architectural elements can be used in the architecture and how they can be interconnected. Thus, although different instances of the same System type are enabled to reconfigure its architecture, they will always maintain type conformance, so that the overall composition is preserved.

6.7.2 Further works

One aspect from our approach that has left to future work, is the updating of the PRISMA Case tool to include the modelling of the reconfiguration aspects as described in this chapter: (i) the graphical modelling of reconfiguration specifications, (ii) the automatic generation of Evolver specifications and the integration of the aspects, and (iii) the graphical depiction of running PRISMA System instances by means of the reactive reconfiguration features provided by Evolver components. However, the principles and knowledge are well established from previous works (Guillén-Martín, 2007), (Pérez et al., 2007), (Pérez et al., 2007a), so its implementation is straightforward.

Another extension to our work is the inclusion of context-awareness features, in order to enable a System instance to react when the system resources decrease or some operating system's signals are raised. This is left as a further work. An initial direction is to use WILDCAT expressions (David & Ledoux, 2005) in the Monitoring aspect to explore the execution context of a System instance, and interact seamlessly with the middleware.

Further works remain, such the dynamic generation of reconfiguration plans from high-level goals. We have used the PRISMA AOADL to define simple event-condition-action (ECA) policies, although other kind of policies could have been defined (Huebscher & McCann, 2008). A promising approach that could be encapsulated inside the ReconfigurationAnalysis aspect are those that carry out task synthesis from high-level goals (Sykes et al., 2008). The contribution of this approach is not the definition of the reconfiguration specification, but the explicit separation between the reconfiguration specifications and the mechanisms that support them. In this way, business logic, reconfiguration specifications, and reconfiguration mechanisms concerns can be maintained separately since they have different rates of change. The business logic can be dynamically changed by the reconfiguration specifications, by means of the reconfiguration mechanisms. And the reconfiguration specifications can also be dynamically changed by using the reconfiguration mechanisms, treating them as any other concern of the system, as we state in the following chapter.

6.7.3 Results

The work related to the definition of the autonomic reconfiguration aspects has produced a set of results that are published in the following publications:

- **C. Costa-Soria**, J. Pérez, J.A. Carsí. *An Aspect-Oriented Approach for Supporting Autonomic Reconfiguration of Software Architectures*. Special Issue on Autonomic and Self-Adaptive Systems, Informatica (Slovenia), vol. 35, issue 1, pp. 15-27. February 2011. ISSN 0350-5596.
- **C. Costa-Soria**, J. Pérez, J.A. Carsí, D. Alonso, F. Ortiz, J.A. Pastor. *Reconfiguración Dinámica de Arquitecturas Software Aplicada a la Tolerancia a Fallos*. In proc. of: 3rd International Workshop on Autonomic and Self-Adaptive Systems (WASELF'10). Valencia, Spain, September 2010 *(in Spanish)*.
- **C. Costa-Soria**, J. Pérez, J.A. Carsí. *Handling the Dynamic Reconfiguration of Software Architectures using Aspects*. In proc. of: 13th IEEE European Conference on Software Maintenance and Reengineering (CSMR'09), pp. 263-266. Kaiserslautern, Germany, 2009.
- **C. Costa**, N. Ali, J. Pérez, J.A. Carsí, I. Ramos. *Dynamic Reconfiguration of Software Architectures Through Aspects*. In proc. of: First European Conference on Software Architecture (ECSA'07). Lecture Notes on Computer Science, vol. 4758, pp. 279-283. Springer, Heidelberg, 24-26 September 2007.
- **C. Costa**, J. Pérez, J.A. Carsí. *Hacia la construcción de arquitecturas software dinámicas*. In proc. of: V Jornadas DYNAMICA, pp. 109-120. Valencia, Spain, 23-24 November, 2006 *(in Spanish)*.
- **C. Costa**, J. Pérez, J.A. Carsí. *Hacia la reconfiguración dinámica de arquitecturas software orientadas a aspectos*. In proc. of: IV Taller de Desarrollo de Software Orientado a Aspectos (DSOA'06), junto a XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD'06). Servicio de publicaciones de la Universidad de Extremadura, Informe técnico TR 24/06, pp. 35-40. Sitges, Spain, 3 October 2006 *(in Spanish)*

CHAPTER VII

DYNAMIC EVOLUTION OF ARCHITECTURAL TYPES

7.1 Introduction

Software systems are continuously evolving during their lifetime. Due to the increasing complexity of software systems, they require continuous updates to correct errors and/or reduce defects. Moreover, due to the usage of such systems in changing environments, software systems are periodically changed to introduce new features (that may also introduce new errors and defects). For this reason, evolution should be considered as an intrinsic part of live software systems.

This evolution is generally performed offline: the system is shutdown, and once the modification has been completed, the entire system is restarted to reflect the new changes. However, this has the disadvantage that the state and pending transactions of the running system are lost. The nature of some systems makes unfeasible this shutdown to integrate changes. This is the case of software systems managing critical resources (e.g. military, avionics, energetic) or that are not directly reachable (e.g. autonomous robots in space explorations). In such cases, runtime evolution support is needed.

The previous chapter describes how autonomic reconfigurations allow a system to proactively adapt or reorganize its structure, and how this can be used to build self-managed systems. However, autonomic reconfigurations only cover one level of architectural dynamism: the dynamic modification of the system topology. Since autonomic reconfigurations operate at the configuration level, they can only change the way that previously defined building blocks (i.e. component and connector types) are instantiated and interconnected. However, the behaviour of these predefined building blocks cannot be changed at runtime; neither new ones can be built at runtime. What happens then if the behaviours of some components or several proactive reconfiguration specifications need to be changed without the system being stopped? In this regard, self-managed systems are not fully dynamic.

To build really open systems, and/or support the updating of highly available systems, an additional level of architectural dynamism is needed: the support for the **Dynamic Evolution of Architectural Types**. This level of dynamism covers the dynamic modification of the types that define the infrastructure and behaviour of a software system. This allows us to entirely modify the architecture of a running system, and thus develop a real *new* architecture at runtime.

This chapter presents an *asynchronous reflective approach* for dynamically evolving architectural types and instances *in a decentralized way*. First, this approach is reflective because it dynamically provides editable specifications of the type to evolve: changes on these specifications are reflected on both the type and their instances at runtime. Second, this approach is asynchronous because the changes on the instances are not immediately applied; each instance is evolved at different times, according to its context. Third, this approach is decentralized because each architectural type can be evolved independently from other types and its instances. Type compatibility is preserved by means of version management, whereas instances are incrementally and asynchronously evolved only after each one is ready to evolve. The approach is presented from a platform-independent view, by describing the different concerns of the dynamic evolution process and how they interrelate which each other.

This chapter is organized as follows. Section 7.2 presents the definitions of type, instance, architectural type, and dynamic evolution of architectural types. These concepts are important since they are used widely thorough the chapter. Next, section 7.3 presents the reflective asynchronous type evolution model and the evolution infrastructure that supports this model. Finally, section 7.4 presents the conclusions and further works.

7.2 Basis of the dynamic evolution of architectural types

7.2.1 Definitions of Type, Instance and Architectural Type

Before describing what the dynamic evolution of architectural types is, first the concept of types and instances should be introduced, to avoid any misunderstanding of the concepts that will be used later.

> *A* **type** *is an abstract concept which defines the* structure *(i.e. the state) and* behaviour *(i.e. how this state is modified) of a software artefact. A type is comprised of two elements:*
>
> *(i)* specification: *the high-level description of a software artefact, and*
>
> *(ii)* executable code: *the realization of this software artefact (i.e. the implementation of the specification).*

In a Model-Driven Development approach (Beydeda et al., 2005) (Selic, 2003), the *executable code* is automatically generated from a (possibly partial and formal) *specification*. This reflects how *specification* and *executable code* are closely related elements. The executable code allows the creation and execution of different instances of the software artefact:

> *An* **instance** *is the execution of a type on a concrete platform: it behaves as defined by the type specification, and is characterized by an internal state (i.e. the data stored in the instance) that is different from other instances of the same type.*

The terms *type* and *instance* are used at different levels of granularity, comprising different kinds of elements, but with the same meaning: the specification of a software artefact and the execution of this specification, respectively. For instance, in object-oriented approaches (i.e. fine granularity levels) types are *classes* and instances are *objects*. Likewise, in component-based approaches (i.e. medium granularity levels) types are *component specititications*, and instances are *component instances*. A component specification can be internally defined as a composition of several classes, whereas a component instance may comprise the instantiation of several objects.

Since in software architectures there are not only components, but also connectors and systems (i.e. composite components), we use the more general terms *architectural types* and *architectural instances*:

> *An* **architectural type** *defines the internal structure and behaviour of an architectural element: a component, a connector, or a system (i.e. a composite component).*
>
> *An* **architectural instance** *is the execution of an architectural type on a concrete platform.*

The specification of an architectural type may be structural or behavioural. The specification is structural if it defines the internal composition of a system, that is, it defines how several architectural types are composed, interconnected, and instantiated. The specification is behavioural if it defines the internal elements that define the behaviour of a component or a connector (e.g. classes, aspects, procedures, etc.).

Moreover, the term *architectural* also provides an additional property: architectural types and architectural instances are coarse granularity elements which only exhibit a set of ports to interact with other architectural elements. They are internally composed of several smaller granularity elements (e.g. classes and objects), but they are not visible from outside; only their relationships with other architectural elements are visible.

7.2.2 Definition of the Dynamic Evolution of Architectural Types

When a type that is deployed and active in a software system (i.e. it has been instantiated), needs to be changed or updated, *dynamic software evolution* is used. Dynamic software evolution is a feature that allows changing a type without the need to shut down the system.

In the context of software architectures, dynamic evolution is performed on architectural types:

> *The* **Dynamic Evolution of Architectural Types** *allows the completely unanticipated runtime modification of the architectural types that a software architecture is build from. This modification comprises the integration of new architectural types, the modification of instantiated types, or the removal of existing types and their instances, while the system is running.*

This enables the modification of architectural patterns, if composite architectural types are changed (i.e. Systems), and the modification of behaviours, if simple architectural types are changed (i.e. Components and Connectors).

It is important to note that the dynamic evolution of a type does not only involve the change of its compiled code, but also the migration (or evolution) of its instances to the structure defined by the new type specification. This is performed by the following evolution process:

1) The specification of the type is changed;
2) The executable code of the type is updated or regenerated, so that new instances could be created according to the updated type; and
3) The current running instances of the type (which are stateful) are evolved or migrated to integrate the changes performed on the type.

The last step is the longest, because it entails the *safe stopping* (see section 3.4) of all the instances that are going to evolve. This guarantees that there are no pending or running transactions which could be affected by the evolution process.

7.3 Reflective Asynchronous Evolution of Architectural Types

This section describes the approach for supporting the dynamic evolution of architectural types. This approach is characterized by the following features:

Type-oriented evolution

The approach focuses on type-oriented evolutions rather than on system-oriented evolutions. This means that evolution support is not considered as a feature of the whole system, but a feature of each architectural type. That is, a software system is not evolvable by itself: evolvable are the architectural types it is made of. Thus, only the architectural types that require dynamic evolution support are provided with evolution mechanisms, whereas non evolvable types not. This benefits the overall performance of the system, since only the evolvable types would be affected by the execution overhead introduced by evolution mechanisms. Another benefit is that it avoids the need of a centralized evolution manager, which could be a single-point of failure. Each evolvable earchitectural type is provided with its own evolution mechanisms, which makes its evolution independent from others.

Reflective evolution description

The approach is reflective because each evolvable architectural type is able to: (i) reify an editable description of itself, and (ii) reflect the changes on this description to itself. The reification operation provides a complete description of the type (in terms of PRISMA elements) that can be obtained at runtime, inspected, and edited through a set of evolution services. The reflection operation performs changes at both the meta-level and the base-level: the specification and executable code of the type is changed, and all its running instances are changed accordingly. The benefit of using a reflective model is that it makes architectural types self-described and self-evolvable. Another benefit is that it simplifies the evolution model from the point of view of the user: he/she obtains the description of an architectural type, he/she edits it, and the architectural type changes itself according to the edited description.

Asynchronous evolution model

The approach implements an asynchronous evolution model because each architectural type and its instances evolve independently, at different rates.

Therefore, each architectural type can process and integrate new evolution requests despite its instances may be still evolving to previous versions. The main benefits of using an asynchronous evolution model are that it enables the development of highly flexible systems, and that it supports the propagation of changes in distributed and/or mobile systems. This has been supported by distributing the evolution mechanisms among the type-level and the instance-level. This is a contribution of this work, since related works have only considered type-level evolution mechanisms.

Instance evolution through transformations

Another contribution of this work is that the evolution of instances has been addressed from a different perspective: through their structural decomposition and transformation. The instance is decomposed in smaller parts and the dependencies among them. Then, changes are performed as a set of incremental changes on these smaller parts, taking into account the dependencies among them. The benefit is that this enables the transparent evolution of instances: changes are performed inside the original boundaries of the instance, without the need of recreating again the external links, or migrating the state of the instance. Only the smaller parts that have been changed may require the updating of links or state migration. This approach is better suited to partially change large architectural types.

These characteristics are described in detail in the following subsections. The subsection 7.3.1 discusses why evolution support is provided at the type-level instead of at the system-level. Then, subsection 7.3.2 describes in detail the reflective model that is provided to evolvable types, and how dynamic changes are introduced. Next, subsection 7.3.3 describes the transformation approach that has been followed to evolve architectural instances. Subsection 7.3.4 describes the asynchronous model that has been followed to support the evolution of types and instances. Finally, subsection 7.3.5 presents the evolution infrastructure that provides support to the characteristics described.

7.3.1 System-level evolutions vs type-level evolutions

Dynamic evolution is generally considered as a *system-level feature*. That is, dynamic evolution support is globally provided, so *all* the types of a system can be subject to dynamic changes (although with some restrictions). Examples of system-level evolution support are the works of (Malabarba et al., 2000), (Ritzau & Andersson, 2000), (Wang et al., 2006). This evolution support generally introduces an execution overhead into the application which should be taken into account. This overhead may range from a 2% in the approach of (Wang et al., 2006) to a 10% in the approach of (Malabarba et al., 2000).

By considering dynamic evolution as a system-level characteristic, it is assumed that all the elements of a system are evolvable, and that all of them can be evolved the same way. However, this is not always true, particularly in large-sized software systems. Since large-sized systems are generally heterogenous (i.e. integrated by elements from different technologies), it cannot be assumed that all the elements of a system could be managed/evolved by a common set of mechanisms. These mechanisms will be different for each technology used and also for each kind of artefact within the same technology. For instance, Java and .NET technologies provide different ways to evolve classes at runtime: in Java this is done by means of the modification of the Java *Class Loader* (Malabarba et al., 2000), (Wang et al., 2006), whereas in .NET this is done by means of *profiling techniques* (Mikunov, 2003). And, within the same technology, different mechanisms are used depending of the kind of artifact to evolve: modification of function pointers to change methods, dynamic linking and redirections to change classes, wrappers to change components, etc. In addition, not all the elements of a system require dynamic evolution support. If dynamic evolution is globally provided, non evolvable elements would be also penalized with the overhead of the evolution infrastructure, despite not using this infrastructure. Only evolvable elements should assume this overhead.

For these reasons, in our approach **dynamic evolution has been considered as a feature provided by the types of a system**, not a feature of the system itself. That is, a software system is not evolvable by itself: evolvable are the types it is made of. Thus, only the types that require dynamic evolution support are provided with evolution mechanisms (and their corresponding execution overhead), whereas non evolvable types not. These evolution mechanisms provide each evolvable type with customised code-generation and state-migration functions to evolve the type specification and its instances.

Furthermore, the consideration of dynamic evolution as a type feature could help to increase the abstraction-level of changes. System-level evolution mechanisms are focused on a specific granularity level: e.g. the evolution of methods *or* classes *or* components. The evolutions on other granularity levels are then done manually. For instance, if dynamic evolution of classes is supported, the evolution of components (considered as compositions of classes) requires manually disassembling a component into its constituent classes, update the required classes and reassemble together the different classes. The complexity increases as higher the granularity level of the type to evolve is from the granularity level of the evolution mechanisms provided. By the contrary, if each type is considered as the provider of its dynamic evolution, it can abstract finer granularity changes and make easy its evolution. Evolutions are requested at a high granularity level (e.g.

component-level), which are internally propagated to the constituent parts (e.g. classes) by requesting finer granularity changes. For instance, evolvable composite components may provide services to dynamically change their internal composition: services to add, replace or remove the internal components it is made of and their relationships. Among these internal components, those that are dynamically evolvable would provide services to change their internal composition: i.e. services to change their internal classes. This is the perspective that has been considered in our approach: evolvable architectural types provide services to change their specification (i.e. its internal composition) and their instances.

7.3.2 A reflective model for evolvable types

In our approach, dynamic evolution is a feature only exhibited by those architectural types that are evolvable. An *evolvable type* is characterised by an evolution infrastructure that allows us to change both its specification and its instances at runtime. This evolution infrastructure is based on the concepts and techniques from the area of *Computational Reflection*.

7.3.2.1 Reflection: The Abstract Model

Computational Reflection (see section 3.5.3) addresses the capability of a software system to reason about itself and act upon itself. In order to do so, a system must have a representation of itself that is *editable* and *causally connected to itself*. The changes that are made in this representation (which is managed as data) are reflected on the system, and vice versa. These systems are called *reflective systems*, and are structured in two levels: a base-level and a meta-level. The *base-level* is the level where the system normally runs, carrying the main functionality it was designed for. The *meta-level* is the level where the editable representation of the system resides. This level allows the system to change its behaviour by modifying its representation. The process of obtaining an editable representation of the system (or accessing the system meta-level) is called *reification*, and the opposite process is called *reflection*.

These concepts have been widely applied in the literature to address the dynamic evolution of software systems. Some of these approaches (see sections 3.5.3, 4.2.1.3 and 4.2.3) provide reflective capabilities at a system-level. The entire software system is structured into a base-level and a meta-level, and *reification-reflection* processes affect the entire system as a whole.

Since our approach focuses on providing dynamic evolution support individually to each type rather than system-wide, our proposal is that *a software system is not reflective: reflective are its evolvable architectural types*. In this way, each evolvable type is structured into a base-level and a meta-level (see Figure 7.1).

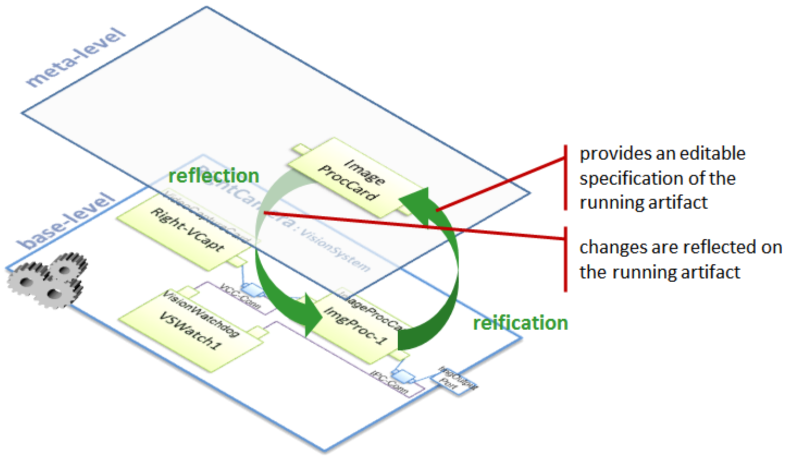


Figure 7.1. Reflective Evolution of Architectural Types

The base-level contains the executable code of an architectural type and its instances. The meta-level contains the specification of the type (i.e. its representation), which is editable and causally connected to the base-level. This (editable) specification is dynamically obtained when a *reification* is requested to an (evolvable) architectural type. A reification is an operation initiated at the base-level which provides access to the meta-level (see reification link in Figure 7.1). All the changes performed on this specification are *reflected* back to the base-level, i.e. they are propagated to the executable code of the architectural type and its instances.

This is the abstract model that has been used to address the evolution of architectural types. However, the realization of this abstract model poses some questions. Since a type does not have a real entity at runtime, how its representation can be obtained and manipulated at runtime? That is, how the meta-level of an evolvable type can be accessed at runtime?

7.3.2.2 Type Meta-Instances: The Concrete Model

To allow the inspection and manipulation of a (evolvable) type at runtime, this type must be made explicit and accessible at runtime. This is performed by **type meta-instances**:

> A type meta-instance *is a type that is materialized at runtime with facilities to inspect and manipulate its meta-level representation.*

In other words, a *type meta-instance* is the objectification of a type at the instance-level, thus allowing base-level entities (i.e. any instance) to access the type meta-level. A type meta-instance provides two services to access this meta-level: *reify* and *reflect*. The *reify* service returns a representation of the type (i.e. a meta-level artefact) that can be inspected and/or manipulated. The *reflect* service takes a representation of the type and uses it to change accordingly the base-level of the type. That is, it changes the executable code of the type and its respective instances.

Internally, a type meta-instance encapsulates the elements that a type is comprised of: the high-level specification, the executable code (which comprises the instantiation mechanisms and the common executable behaviour of instances), and the code generation patterns that transform the specification into executable code. This allows the meta-instance to build the meta-level representation of the type, so other instances can inspect and manipulate it, and to regenerate the type base-level (i.e. the executable code of the type) each time the meta-level representation is edited.

Figure 7.2 describes graphically the different dimensions where a type meta-instance acts: the type-level, the instance-level (both at the base-level), and the meta-level.

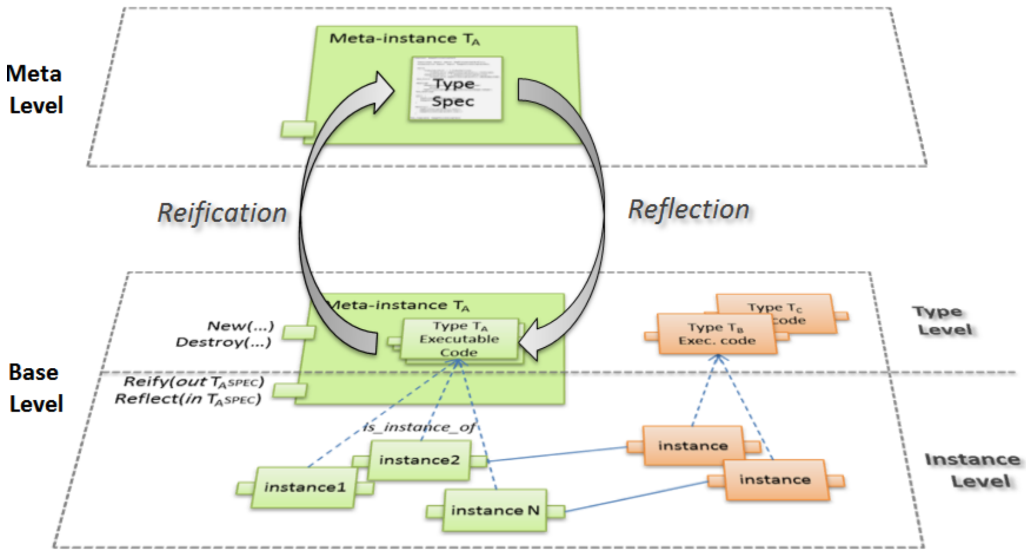


Figure 7.2. Model of Type Meta-Instances

A type meta-instance, *Meta-instance* T_A, behaves as the type it materializes (T_A), because it encapsulates the executable code of this type (see the element called *Type* T_A *Executable Code* in Figure 7.2), thus providing the mechanisms to create and destroy instances (see the port publishing the services *New* and *Destroy*). For this reason, a meta-instance acts at the type-level. On the other hand, a type meta-instance is accessible by other instances, to allow them to invoke meta-level services (i.e. see the port publishing the services *Reify* and *Reflect* in Figure 7.2). For this reason, a meta-instance acts at the instance-level. Finally, a type meta-instance contains the reification of the type it materializes (see the meta-level element *Type Spec* in Figure 7.2) and is able to reflect the changes on this reification to the base-level. For this reason, a meta-instance acts at the meta-level.

According to the taxonomy of reflective models presented by Walter Cazzola in (Cazzola, 1998), our approach could be categorized as a *Meta-Class Model*: the reflective tower is realized by the instantiation link among a type and its instances. That is, evolvable types, by means of type meta-instances, provide reflective capabilities to reify and manipulate its meta-level representation. The changes on this representation are propagated to the base-level by manipulating the instantiation link among a type and its instances. When a new type version is generated, old instances are migrated from the old type version to the new type version. This results in a migration of the instantiation link.

7.3.2.3 Reification of Types

In our approach, the reflective model presented in the previous section is used to evolve both simple and composite architectural types. Externally, type meta-instances materializing simple and composite types behave identically: they create instances of the type that they materialize and they provide meta-level services (i.e. *Reify* and *Reflect*) to obtain a reification of the type and change its base-level. However, meta-instances of simple types differ from meta-instances of composite types in that the reification provided is completely different: the definition of simple architectural types is different from composite architectural types. For instance, in PRISMA, simple architectural types (i.e. Components and Connectors) are defined by means of *aspects*, *weavings* and *ports*, whereas composite architectural types (i.e. Systems) are defined by means of *architectural types*, *attachments*, *bindings*, and *ports*.

To abstract from these differences and provide a unified view for evolving types, reifications are provided through *<Type>Spec* objects. The term *<Type>Spec* is an abstraction for referring to the reification of an architectural type independently of what kind of architectural element (i.e. simple or composite) this reification represents. For instance, in case of simple type reifications, this object is called *SimpleSpec*; in case of composite type reifications, this object is called *CompositeSpec*.

A *<Type>Spec* object (i.e. a *SimpleSpec* or a *CompositeSpec*) is returned by the *Reify* service of type meta-instances: it encapsulates the reification of a type through a set of data structures that can be inspected and manipulated at runtime. To guarantee that the values contained in the data structures are changed consistently and that the resulting type reification is syntactically correct (i.e. according to the metamodel), the *<Type>Spec* object provides a set of services to edit these data structures. These services are called *evolution services*, because they edit the reification in terms of *additions*, *removals* or *replacements* of elements.

For instance, Figure 7.3 shows the structure of a *SimpleSpec* object (see the class called *ComponentSpec*). A *SimpleSpec* object has a set of data structures for describing the elements that a simple PRISMA architectural type is composed of: (i) aspects (defined in *AspectSpec* data structures), (ii) ports (defined in *PortSpec* data structures), (iii) weavings (encoded in *WeavingSpec* structures), and (iv) a constructor (defined through the data structures *ParamInfo*, which defines the constructor parameters, and *ExpressionSpec*, which defines the constructor code).

Note that the reification of simple architectural types only contains the name of subtypes, such as aspects, but not the reification of such subtypes (see in Figure 7.3 the attribute *aspectType* in *AspectSpec*). This is because **our reflective model is recursive**: *the reification of subtypes* (e.g. aspects) *must be requested to their respective meta-instances*. This improves maintenance through the delegation of tasks, and the makes the model simpler.

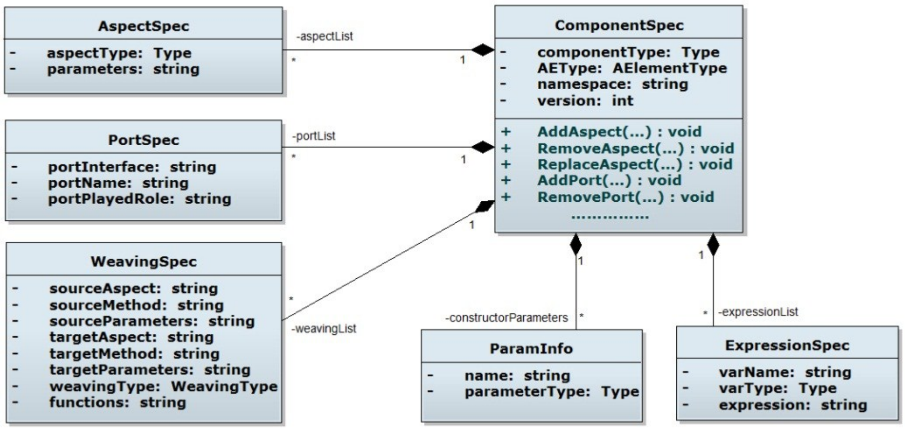


Figure 7.3. Structure of a *SimpleSpec* object

The edition of the values contained in *SimpleSpec* objects is done (and validated) through a set of evolution services: *addAspect*, *addPort*, *addWeaving*, *removeAspect*, *removePort*, *removeWeaving*, etc. These services validate that the changes introduced are syntactically conformant to the metamodel. Examples of these validations are the following:

- *AddAspect*: an aspect can be only added if there is no another aspect of the same type already defined in the type;
- *RemoveAspect:* an aspect cannot be removed if there is a port or weaving that refers to this aspect;
- *ReplaceAspect:* an aspect can only replace another if: (i) the old aspect exists, (ii) the new aspect does not exist previously, (iii) in case a port exists that publishes interfaces from the old aspect, the new aspect implements these interfaces, and (iv) in case a weaving exist with the old aspect, the new aspect must provide the same service intercepted by this weaving
- *AddPort*: a port can be only added if: (i) there is no another port with the same name, and (ii) there is an aspect that implements the interface and PlayedRole that the port defines.
- *AddWeaving:* a weaving can be only added if: (i) the aspects it synchronizes are defined in the type, and (ii) the services it intercepts are provided by each aspect.

In this way, through the use of evolution services, *<Type>Spec* objects allow base-level elements (i.e. computational entities) to inspect and manipulate type reifications as data elements without the risk of introducing inconsistent changes.

In addition, another advantage is that *<Type>Spec* objects facilitate the definition of sets of changes, as it is described in the next section.

7.3.2.4 Evolution Process Overview

The dynamic evolution of a type is reactively performed. That is, the definition of the changes to carry out on a type is performed externally. The entity that defines the set of changes to apply on a type and that initiates the evolution process is called **evolution agent**. This *evolution agent* can be a human (which performs ad-hoc evolutions), or an architectural element (which performs programmed evolutions).

To dynamically evolve a type, an evolution agent must follow three steps: (i) reify the type, (ii) manipulate the reification provided, and (iii) reflect the changes to the meta-level. These steps are graphically described in Figure 7.4.

The first step is to obtain the reification of the type to inspect and/or evolve. This reification is obtained by invoking the *Reify* service provided by the meta-instance of the type to evolve (if the type is not evolvable, then its meta-instance will not be available). The *Reify* service returns an object *<Type>Spec* (i.e. a *SimpleSpec* or a *CompositeSpec* object, depending of the kind of architectural type reified). This is described in Figure 7.4 as the step 1. Note that, in the figure, the *Reify* service returns a *SimpleSpec* object, named as C_{SPEC}. This is because the type T_A materialized by the meta-instance is a simple architectural element. Recall that externally, simple and composite architectural elements are black boxes which cannot be distinguished from each other. This distinction can be only made when inspecting the internal structure of a type, which can be only performed by obtaining its reification.

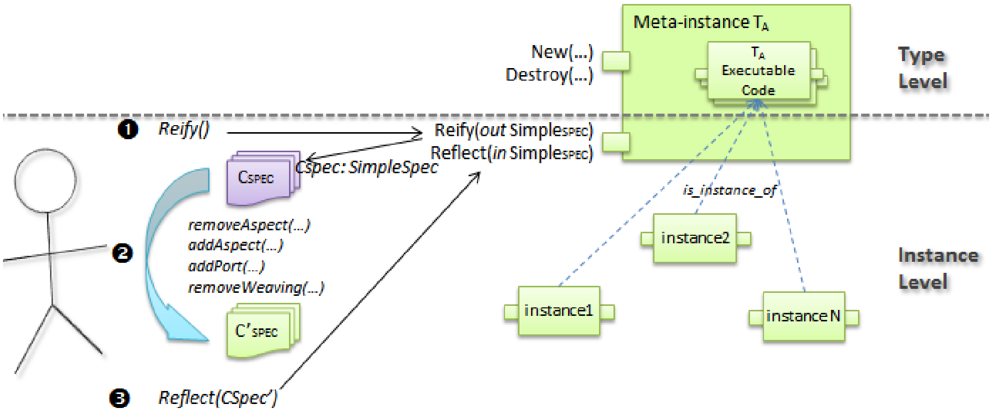


Figure 7.4. Black-box view of the dynamic evolution process

The second step is to define the set of changes that are required on the type. This is performed by means of the specific evolution services that *<Type>Spec* objects provide. This is described in Figure 7.4 as the step 2: the evolution agent manipulates the object obtained in the previous step, C_{SPEC}, by using the evolution services this object provides (e.g. *removeAspect*, *addAspect*, etc.). The resulting specification is named in the figure as C'_{SPEC}.

We assume that the set of changes that are going to be introduced in the type are correct and valid. If new functionalities are added to an architectural type, the interacting types must be also appropriately updated to correctly use the new functionality (this also applies when removing functionalities). This is responsibility of the architect. The evolution infrastructure is only responsible for guaranteeing that the changes requested are correctly introduced. This work is only concerned with the correct execution of dynamic changes, but not with trying to establish whether the changes proposed are correct.

Another important aspect to take into account is that either architectural types are self-contained (i.e. they are provided with all the internal types that they require) or their dependencies are made explicit (i.e. they explicitly publish which are the provided and required services or types). This is a fundamental principle for enabling the dynamic evolution process.

Finally, the third step is to reflect the changes on the type meta-level (which results in the dynamic evolution of the type base-level –the modification of the type definition and its instances). This is performed by calling the service *Reflect* provided by the type meta-instance. This service requires a *<Type>Spec* object of the same kind that the meta-instance provides, and that was generated from the current type version. This is described in Figure 7.4 as the step 3: the evolution agent invokes the *Reflect* service, by providing the edited reification, C'$_{SPEC}$. Then, the evolution of the type starts: a new type version will be generated and its old instances will be migrated/transformed to the new type version.

Note that the edition of a type reification is performed asynchronously: the evolution agent does not need to interact with the type meta-instance during the edition; only when it wants to reflect the changes. This gives great flexibility for evolving running artefacts, without making type meta-instances to wait until inspection or manipulation of reifications finish. For instance, an evolution agent may encapsulate a tool that enables a user to manipulate a type description graphically. When the changes on this graphical description finish, they are provided to the type meta-instance and reflected on the type and their instances. Another example is the use of mobile agents: the evolution agent may be transferred over the network to enable remote updating of software artefacts. Thus, a type meta-instance can serve several, concurrent reification requests.

However, *concurrent modifications* of a type (and version branching) is not allowed: for simplicity, our approach has been limited to allow only *a single evolution path* (see page 271). For this reason, in presence of several evolution agents working on the same reification version (i.e. editing a *<Type>Spec* object generated from the same type version), only the first agent to reflect its changes (i.e. call the *Reflect* service) could evolve the type. A new type version would be generated and the other delivered reifications would be invalidated. When the other agents try to reflect their set of changes (which have been made to an invalidated reification), then the meta-instance will notify them that their reifications are outdated and that they should obtain a new reification. This is a simple but yet powerful solution. It does not only guarantees that a single evolution path is built, but also makes evolution agents to take into account the new changes and to consider if additional changes to those initially planned are needed.

7.3.3 Evolving instances through transformations

When a set of changes is reflected to the type base-level, this results in the generation of a new type version and the evolution of the instances of the old type version. Most of current approaches address the evolution of instances through a *migration approach* (e.g. (Vandewoude & Berbers, 2005), (Ritzau & Andersson, 2000)): each instance of the old type version is recreated as an instance of the new type version, including the migration of its previous state to the new instantiation. This recreation is achieved through the following steps:

1) An instance of the new type version is created: it has no previous state, it is initially stopped, and disconnected from other instances.
2) The old instance is stopped, to avoid that its state could change.
3) The state of the old instance is migrated (and possibly adapted) to the new instance (see section 3.4.2).
4) The connections from and to the old instance are changed to point to the new instance.
5) The new instance is started.
6) The old instance is deleted.

This approach has the disadvantage that is not totally transparent to the linked instances: the existing connections must be changed to point to the new instance, and a little disruption is during the time the old instance is stopped and the new instance is started. However, the big disadvantage is that this approach requires an explicit control of the type meta-instance to drive the evolution process. That is, instances do not evolve autonomously, i.e. they require an external management.

In this thesis we propose an alternative approach: to use a **transformation approach based on structural decomposition**. In this approach, the evolution of instances is addressed by means of the transformation of their internal structure to accommodate the changes introduced by the new type version. This transformation is based on the decomposition of the instance structure in smaller parts and dependencies among them. Then, changes are performed as a set of incremental changes on these smaller parts, by adding, removing or replacing them, taking into account the dependencies among these parts.

The result is that an instance is not recreated again, but evolved from inside. From outside, an evolved instance keeps its original boundaries, links and state, but also integrates the behaviour added by the new type. From inside, only the parts that have been affected by the change are modified: their state is migrated to new ones (in case of replacements), whereas the state of non-changed elements is kept intact. Type transformation approaches are better suited to partially change large architectural types (e.g. servers), because it is not required the complete stopping of the entire architectural instance; only the required parts of the instance.

A transformation approach can be only realized if the following two conditions hold:

- The type to evolve can be decomposed into smaller entities (or parts) and the interrelations among these entities.
- Type instantiations (i.e. the executable code) preserve the type structure, so they can be also decomposed in smaller entities.

These conditions hold in the case of PRISMA architectural elements. For instance, PRISMA simple components can be decomposed into entities and interrelations: aspects are entities with an internal state, whereas weavings and ports are interrelations among entities (weavings synchronize aspects, and ports publish inbound and outbound aspect services). Since the implementation of PRISMA simple components (Costa-Soria, 2005), (Pérez et al., 2005a) preserves the internal elements (i.e. aspects, weavings and ports are implemented as classes which are composed to build a component), then PRISMA instances can be evolved through a transformation approach.

In this way, the evolution of instances is performed through the following process:

1) When a new type version is produced, each instance receives the evolutions to apply, as a set of incremental changes on its current structure (i.e. additions, deletions or replacements)
2) The elements that are going to be changed (i.e. the internal parts and their relationships) are isolated from the rest, that is, stopped and unassembled if necessary.
3) The incremental changes are applied on the isolated parts. Replacement of old parts can be performed through migration or through its internal transformation, recursively.
4) The changed parts are reassembled again to the instance.

The main advantage of instance transformation, as opposite to instance migration, is remarkably when the types evolved are composed of concurrent entities which are highly independent among them, such as software architecture specifications (i.e. composite components) and aspect-oriented components (i.e. PRISMA simple components). Another advantage is that the evolution of instances is transparent to other interacting instances, since their boundaries are not modified by the evolution process.

7.3.4 An asynchronous model for types evolution

Dynamic evolution can be performed *synchronously* or *asynchronously*, depending on how a type and its instances evolve with respect to each other.

- In **dynamic synchronous evolution**, *a type and its instances are evolved sequentially*, i.e., the evolution of a type is followed immediately by the update of its instances before the type can evolve any further.
- In **dynamic asynchronous evolution**, *a type and its instances are evolved independently, at different rates*. The type may evolve again before all of its instances have finished the integration of the previous changes.

The advantage of asynchronous evolution is that it supports frequent change requests, but also the (deferred) propagation of such requests to distributed or partly reachable instances. On the one hand, changes can be performed earlier: a type can be evolved as soon as required, without waiting to update all of its instances. This is useful for developing highly flexible systems, i.e. those systems with a high probability of changes. On the other hand, changes can be propagated and applied to each instance at different times, without requiring instances to be permanently reachable or online. This is particularly useful for supporting dynamic changes in distributed systems and/or mobile systems.

For these advantages, our approach has been designed to support an asynchronous evolution model. Next, the synchronous and asynchronous evolution models are described in detail, and after that, our asynchronous evolution approach will be introduced.

7.3.4.1 Modelling evolutions over time

Evolution involves change, and change in turn involves time: the instant when a change is produced. This time is important, as it determines the point from when evolutions are materialized in a system. This may involve the coexistence of instances from both the original type and the evolved type.

To distinguish among the different evolutions of a type over time and keep track of them, type evolutions are captured as **type versions**. Each time a type is dynamically evolved, a new type version is dynamically created/generated. This version contains the new (evolved) type specification and the executable code from which new instances will be created.

A type T evolves from a type version T_i (where $i \in \mathbb{N}$, and i=0 is the initial type version) to the next, T_{i+1}, via an **evolution process**. An evolution process starts when a new *evolution request* is received, and involves: (i) a type-level action, which changes the current type version and generates a new one; and (ii) instance-level actions, to migrate/evolve the existing instances to the new type version, preserving their internal state. An evolution process ends when all the instances have been evolved[33]. Depending on the evolution model, evolution processes (and type versions) may overlap over time (in asynchronous models) or not (in synchronous models).

An abstract, illustrative example

To illustrate these concepts, and how they relate to synchronous and asynchronous evolution models, a graphical example is used. Figure 7.5 describes visually how a type T and its instances evolve over time. It captures the time when a new type version, T_1, is introduced in the system and when the old instances are evolved to the new version.

In this figure, the vertical axis depicts the different evolutions, or *type versions*, of the type T, whereas the horizontal axis depicts the time when some actions (i.e. evolutions or instantiations) take place. Type versions are depicted as squares with the number of version they represent (e.g. T_0, T_1). The position of each square represents the time when the corresponding type version was introduced in the system, and the circles represent its instantiations (which are identified as unique numbers).

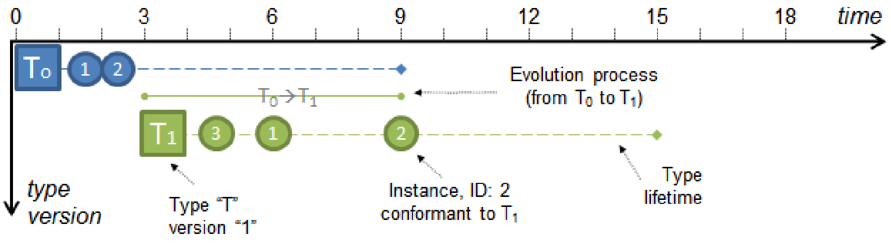


Figure 7.5. Capturing evolution over time

It is important to note that an instance belongs to (*is conformant to*) a certain type version at a certain time, but later it may belong to a different type version (i.e. after evolving to a new type version). That is, an instance keeps its identity (captured by its internal state) among evolutions. This fact is also captured in the figure: circles with identical number represent the same

[33] For the sake of simplicity, it is temporarily assumed that *all* the existing instances must integrate the new changes. However, this assumption is not realistic, since it does not always hold. In some cases, some instances should remain unevolved when there are compatibility issues of the new version with other existing types. This has been taken into account in our approach, but will be considered later. The key idea here is that an evolution process *ends when it finishes evolving the instances that it has been requested to.*

instance, whereas their colour represent the type version to which they are conformant to.

The figure depicts that the type T is originally introduced at instant 0 (and materialized as the type version T_0), and that two instantiations are created at time instants 1 and 2, respectively. Then, at time instant 3 a new *evolution process* starts, which: (i) introduces/generates a new type version, T_1; and (ii) evolves the existing T_0 instantiations (i.e. instances 1 and 2) to the structure of the new type version. Evolution processes are depicted as solid round-ended lines, tagged with the name of the type version to evolve and the name of the new type version. The evolution process "$T_0 \rightarrow T_1$" starts at time 3 and finishes at time 9, when all the instances of T_0 have been evolved: instance 1 evolves at time 6 and instance 2 at time 9.

Figure 7.5 does not capture the reasons motivating instance evolution delays. These reasons may be that: (i) the instance is waiting to reach a quiescent status, (ii) the instance cannot be evolved due to compatibility issues with other interacting instances, or (iii) the instance has not received yet the evolution request (e.g. in distributed systems).

Note that a new type version can be instantiated as soon as it is introduced in the system, although the evolution process that has introduced this type version has not finished yet. For instance, see how the instance 3 is created at time 5 while the evolution process $T_0 \rightarrow T_1$ is still running.

When addressing dynamic evolution, most of the approaches from the literature have only considered scenarios with a single evolutionary step, as the described in the previous example. However, in these scenarios the advantages of the asynchronous evolution model cannot be perceived, since it behaves identically as in the synchronous evolution model. It is only when considering several evolutionary steps (three or four consecutive evolution requests) when the differences among the synchronous and asynchronous evolution models emerge.

Next, this example is extended with additional evolutionary steps, which will better illustrate the differences among the synchronous and asynchronous evolution models.

Synchronous Evolution Model

In the synchronous evolution model, as implemented in most of current approaches (e.g. (Nicoara et al., 2008), (Wang et al., 2006)), the main limitation is that a new evolution process cannot be started until the completion of the previous evolution process. In other words, *in the synchronous evolution model, evolution processes cannot overlap each other*. This is because in this model, an evolution process replaces the old type version with the new version. And since instances are linked to a type version, this replacement cannot be effectively completed until all the old instances have been evolved to the new type version. As a result, new evolution requests must be delayed until all the existing instances have integrated the previous evolution request.

Figure 7.6 shows how the type T used in the previous example, when receiving two additional evolution requests, is evolved following a synchronous model. These evolution requests are produced at time instants 6 and 11, which will initiate two evolution processes, called $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_3$, respectively. However, these evolution requests must be delayed to avoid the overlapping of evolution processes, not supported in synchronous models.

For instance, according to the figure, the evolution process $T_1 \rightarrow T_2$ cannot be started until time instant 9, when the evolution process $T_0 \rightarrow T_1$ finishes (which finishes when the last instance of T_0, identified as 2, is evolved to T_1). The delays of evolution processes are illustrated in Figure 7.6 as a dotted line at the beginning of each evolution process. The figure also shows how the delay of this evolution process results in the delay of the following evolution process, $T_2 \rightarrow T_3$, which will be propagated consequently over time if the number of instances to evolve is significant.

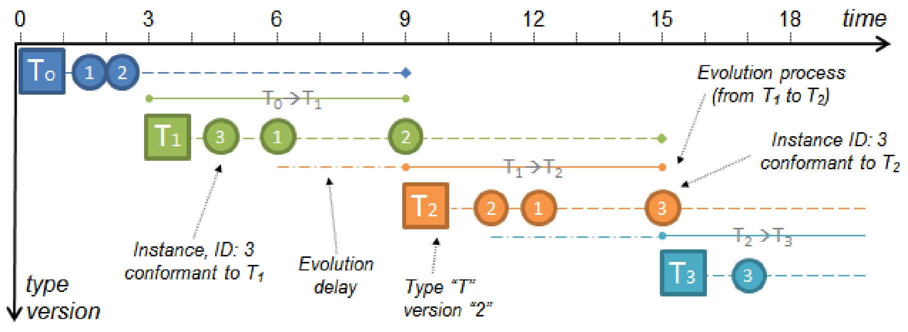

0
3
6
9
12
15
18
time
T0
1
2
T0→T1
Evolution process (from T1 to T2)
T1
3
1
2
Instance ID: 3 conformant to T2
T1→T2
T2
2
1
3
T2→T3
T3
3
Instance, ID: 3 conformant to T1
Evolution delay
Type "T" version "2"
type version

Figure 7.6. Synchronous evolution model

As it can be observed, the main disadvantage of synchronous evolution is the time that is needed until a new change can be introduced in a system. This limitation is mainly due to that type and instances evolve synchronously: when a type is changed, their instances must be adapted before a new change request could be processed.

This may be an important drawback when evolving either distributed or mobile systems, where network fluctuations and/or the reachability of the instances (e.g. some may be disconnected) may increase the time needed to propagate new changes, and thus, the time needed to perform several dynamic changes.

Asynchronous Evolution Model

In the asynchronous evolution model, types and instances evolve at different times. This allows evolution processes to overlap their execution: as soon as a type has been evolved, it can be evolved again, although some of its instances may still be applying the previous changes. In this case, the delay produced to evolution processes, as in the synchronous evolution model, disappears[34].

Using the same example presented in Figure 7.5, next it is illustrated how the type T is evolved following an asynchronous evolution model. As in the synchronous case, two additional evolution requests are received at time instants 6 and 11, which will initiate two evolution processes, called $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_3$, respectively.

Figure 7.7 shows how these evolution processes are managed in an asynchronous evolution model: they are started immediately, as soon as the evolution requests are received (at time instants 6 and 11, respectively). This is done although other evolution processes are still active. For instance, at time instant 7, two evolution processes are active, $T_o \rightarrow T_1$ and $T_1 \rightarrow T_2$. The former is evolving instance 2 from T_0 to T_1, whereas the latter is evolving instances from T_1 to T_2.

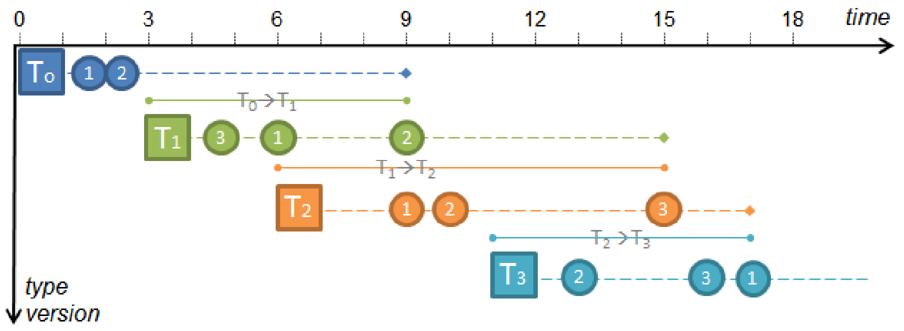


Figure 7.7. Asynchronous evolution model

The overlap of evolution processes implies the coexistence of several type versions and instances belonging to different type versions. For instance, at time instant 13, two evolution processes are active: $T_1 \rightarrow T_2$ and $T_2 \rightarrow T_3$. The former is evolving instance *3* from version T_1 to version T_2, and the latter is

[34] The only delay that is still present and cannot be removed is that produced by the instances to evolve to the next type version, due to their evolution autonomy.

evolving instance *1* from version T_2 to version T_3. This also means that three type versions are coexisting simultaneously: T_1, T_2, and T_3, providing the behaviour of instances *3*, *1*, and *2*, respectively.

This is a consequence of a feature of the asynchronous evolution model: each instance can evolve independently of the other instances, without waiting for the other instances to finish the previous evolution process(es). As soon as an instance is ready to evolve to the next type version, it does, although other instances remain in *k*-previous versions. In order to support this, an adequate version management must be considered.

7.3.4.2 Additional characteristics

The approach presented in this thesis provides support to the asynchronous model presented above. In addition, this approach takes the following considerations: evolutions are performed incrementally, by following a single evolution path, and unloading the inactive type versions to preserve system resources. These considerations are further explained below.

Incremental evolutions

Our approach is focused on an incremental evolution setting: the set of changes performed at time *t* (both at type-level and instance-level) is performed on the result of changes performed at time *t-1*. This is an intrinsic property of evolution: changes are performed building from previous experience. This also benefits the evolution process: it is not needed to completely describe the new type version, but also the differences with respect the previous type version. This reduces the size of data that must be transferred to communicate changes among types and instances, and also allows changing instances through transformation instead of through migrations (see section 7.3.3).

Another advantage of incremental evolutions is that it allows some instances remain unevolved, or disconnected from evolution, for some time. If later these instances require to integrate the latest changes, they simply would need to integrate the set of incremental changes that have been produced from their outdated version to the most recent type version.

Single Evolution Path

While in a synchronous evolution model only one evolution process is active (thus only the old type version and the new one coexist temporarily), in an asynchronous evolution model several versions and evolution processes can coexist at the same time. However, this does not necessarily mean that several *evolution paths* (i.e. version branching) are allowed. The support for multiple evolution paths poses several problems that are outside the scope of this work, so they are avoided in our approach.

A single evolution path has been guaranteed by constraining the evolution to only the latest type version, protecting it from concurrent evolution requests. This is done by adding two restrictions to evolution processes:

- A new evolution process cannot be started until the previous one has generated at least the new type version.

  That is, it is introduced a delay equivalent to the time needed to generate and introduce a new type version in the system. However, this is a logical constraint, since it makes no sense to evolve something (from an incremental perspective) that has not been produced yet.

- An evolution process cannot be finished if a previous evolution process is still active.

  This is because the previous evolution process may still be evolving instances, which must be also evolved by the newer evolution process in order to integrate the newer changes.

At the end, all the instances must follow the evolution path built over time, that is, to integrate all the sequence of changes (i.e. evolution requests) received. Only in this case we would be able to determine the correct version an instance must be evolved to, and adequately manage the different type versions.

Activeness of type versions

Since an evolution path may consist of several type versions and not all of them would be used, to save computer resources the concept of *activeness of type versions* has been used.

A type version is **active** if: (i) it is the most recent type version, or (ii) it has running instances. While a type version is active, it is kept in memory; otherwise, it is unloaded from memory, freeing its resources.

The most recent type version is active by default to allow its evolution: only the latest version can be evolved. The previous versions are outdated and are only kept in the system to allow the execution of out-of-date instances. As soon as out-of-date instances have been updated or removed, then the old type version is not active anymore and can be safely unloaded from the system. In general, the end of an evolution process implies the inactivity of the type version that has been updated, due to the evolution of all of its instances to the new type version.

It is important to note that *version inactivity is considered only within the context where this type has been imported to* (e.g. a composite component or System). Every System integrates a local copy of a type, and each local copy may integrate the different versions (i.e. evolutions) at different times. A type version may be inactive in one System (because it has been successfully evolved), but still be active in another System (because it has local instances pending to update). For this reason, the accessibility of inactive versions must be guaranteed, by storing them in a permanent storage for future reference, such as a database or the filesystem.

7.3.5 Description of the evolution infrastructure

This section describes the infrastructure that supports the asynchronous dynamic evolution of architectural types. This infrastructure is integrated into each evolvable architectural type, and is distributed among meta-level artefacts (i.e. type meta-instances) and base-level artefacts (i.e. instances). This distribution is one of the fundamental points that make possible the asynchronous evolution of a type and its instances: to allow this kind of evolution[35], the type-level and the instance-level must be separately managed.

For this reason, **meta-instances perform type-level operations,** whereas **each instance performs instance-level operations**. On the one hand, a meta-instance manages the evolution of the specification of a type, the executable code of this type, and its different versions. In this way, a meta-instance can address several evolutions of the type independently of the state of its instances. On the other hand, each instance manages how it reaches a safe status for evolution (i.e. quiescence), the transformation of its internal structure, and/or the migration of its previous state to the new version. In this way, each instance may integrate the different evolutions at different times.

According to the PRISMA model presented in section 2.4, there are two kinds of architectural elements: *simple* (i.e. a Component or a Connector) and *composite* (i.e. a System). They differ on how these types are internally composed: a simple type is composed of aspects, weavings and ports; whereas a composite type is composed of ports, other architectural types, and interactions among them (attachments and bindings). The infrastructure for evolving simple and composite types is quite similar; it only differs on the evolution services provided to change the internal composition of the types. For instance, the evolution services for simple types are: *addAspect*, *addWeaving*, *addPort*, etc., whereas the services for composite types are:

[35] Which, as described in section 7.3.4, is characterized by the different rates at which types and instances evolve respect each other.

addArchElement, *addAttachment*, *addBinding*, etc. For simplicity reasons, only the evolution details for simple types are described.

Next, the internal structure of simple type meta-instances and simple instances is described in detail.

7.3.5.1 Type-level Evolution

Type-level evolution, i.e. the dynamic generation of a new type version and the replacement of the old type version, is carried out by type meta-instances. Recall that a type meta-instance is the materialization of a type at runtime, which provides facilities to inspect and manipulate its meta-level representation (see section 7.3.2.2). For each evolvable type defined in PRISMA, a type meta-instance is automatically generated.

The internal structure of a type meta-instance consists of four modules or functional areas:

1) *TypeDescription*, which encapsulates the reification of the type, provides introspection services, and manages the different type versions.
2) *Builder*, which provides the code responsible for the creation and destruction of type instantiations (i.e. the executable code of the type).
3) *TypeEvolution*, which is in charge of reflecting changes made on a type reification to the type-level. It also encapsulates the domain-independent code generation patterns that produce a new type version.
4) *EvolutionMonitoring*, which keeps the connection with the instance-level: manages the population of instances, propagates evolutions to these instances, and supervises the asynchronous evolution.

Type meta-instances have been integrated in PRISMA by using the same concepts that the PRISMA model provides: they are modelled as *Components*, and their behaviour is defined through *aspects*. These aspects encapsulate the functional areas described above, because each functional area implements a different concern of the evolution process, which is shared among other meta-instances. By shared we refer to the fact that the specification of each aspect is common for any simple type meta-instance, differing only in the state they acquire when they are instantiated. This is the case of the aspects *TypeDescription*, *TypeEvolution* and *EvolutionMonitoring*, which are initialised with the data of the type encapsulated by the meta-instance. Only the *Builder* aspect is not reusable, because it is completely different for each type meta-instance: it contains the instantiation process of a specific architectural type.

Figure 7.8 shows the internal structure of a type meta-instance as implemented in PRISMA. The four aspects are synchronised through weavings, which are automatically generated.

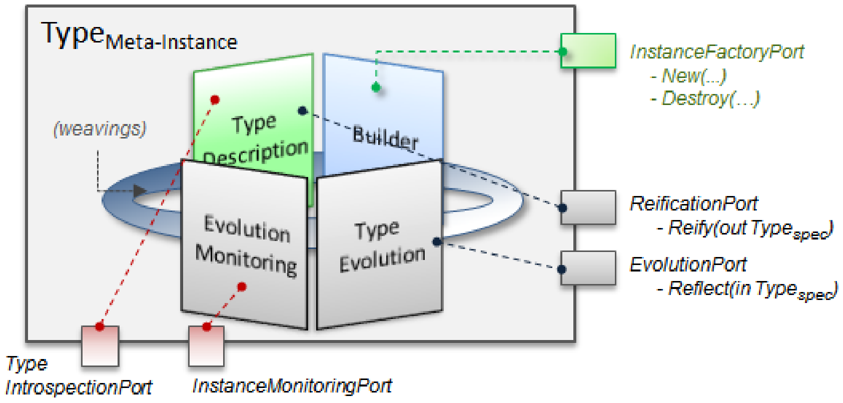


Figure 7.8. Internal structure of a Type Meta-Instance

For instance, the services provided by the *Builder* aspect for creating and destroying instances are weaved to the *EvolutionMonitoring* aspect, to update the population of instances (see Figure 7.9).

```
Weavings
   ...
   EvolutionMonitoring.RegisterInstance( new InstanceInfo(
            instanceRef.ID, instanceRef, instanceRef.version))
      after
   Builder.NewInstance(initParams, output instanceRef);

   EvolutionMonitoring.UnRegisterInstance(instanceRef)
      after
   Builder.DestroyInstance(instanceRef);

   // ... more weavings
End_Weavings;
```

Figure 7.9. Weavings that manage the population of instances

For instance, the first weaving of this figure describes the following behaviour: **after** the successful execution of the *NewInstance* service provided by the *Builder* aspect, the service *RegisterInstance* provided by the *EvolutionMonitoring* aspect must be executed. Note how the output of the *NewInstance* service (i.e. *instanceRef*, the reference of the instance that has been created) is used as a parameter for the *RegisterInstance* service. The other weaving is quite similar: **after** the successful execution of the *DestroyInstance* service, the service

UnRegisterInstance of the *EvolutionMonitoring* aspect must be triggered (which will remove the instance reference from the population list). In this way, explicit references among aspects are avoided, maintaining them loosely coupled, which benefits aspect maintenance and reuse.

The services provided by the aspects are published through a set of ports, which are of two kinds:

- Public ports: These ports are opened to any software artefact of the system, although they are targeted to be used by evolution agents, that is, entities that manipulate evolvable types.
    - *InstanceFactoryPort*: This port publishes the creation and destruction services provided by the *Builder* aspect. It allows creating new instantiations of the current type version.
    - *ReificationPort*: This port publishes the *Reify* service, provided by the *TypeDescription* aspect. It allows the evolution agents to obtain the reification of the current type version.
    - *EvolutionPort*: This port publishes the *Reflect* service, provided by the *TypeEvolution* aspect. It allows the evolution agents to evolve the current type version.
- Internal ports: These ports are for private use among the type meta-instance and the instances it manages.
    - *TypeIntrospectionPort*: This port allows instances to introspect their type. That is, to get information about their current type version, such as constraints, allowed types, data structures, etc.
    - *InstanceMonitoringPort*: This port propagates evolution requests to instances. It is used by the *EvolutionMonitoring* aspect.

Next, each aspect of a simple type meta-instance is described in detail.

Builder Aspect

The *Builder* aspect encapsulates the executable code of the type that: (i) is required to create and destroy instances, and (ii) provides the behaviour of these instances.

This aspect behaves as an instance factory: it provides two services, *New* and *Destroy*, which carry out the actions defined in the *new* and *destroy* sections of a PRISMA type specification, respectively. On the one hand, the service *New* takes as input a set of initialization values (if required by the type), creates a new instantiation of the type, and returns a reference to this instantiation (i.e. an instance ID). On the other hand, the service *Destroy* takes as input the reference of an instance created by this aspect and destroys it.

Since this aspect encapsulates the executable code of the type, it must be updated every time the type evolves. For this reason, every time the type is evolved, the *Builder* aspect is automatically regenerated to reflect the new changes (not only in memory, but also in disk; more details are later explained in page 284). That is, this aspect contains the latest executable type version. Thus, new instances will always be created from the latest type version, favouring the progressive adoption of newest versions.

For illustration purposes, Figure 7.11 and Figure 7.11 show the executable code (in C#/.NET) that is generated from the PRISMA specification of an evolvable type, the *ImageProcCard* type. Two classes are automatically generated: *ImageProcCard* and *ImageProcCardBuilder*.

The *ImageProcCard* class (see Figure 7.11) is the materialization of the *ImageProcCard* type in .NET. This class behaves as a PRISMA Component: it inherits its behaviour from the class *ComponentBase* and is conformant to the *IComponent* interface. The constructor and destructor of this class (i.e. the methods *ImageProcCard*[36] and *Destroy*, respectively) make use of the generic behaviour provided by the PRISMA model, through the invocation of *base* methods (i.e. the inherited constructors and destructors of PRISMA Components).

```
public class ImageProcCard : ComponentBase, IComponent {
   static IMetaComponent meta;

   public ImageProcCard(string name, params object[] parameters):
          base (name) {
         meta.BuildInstance(this, parameters);
   }

   public void Destroy() {
      meta.DestroyInstance(this);
      base.Dispose();
   }
}
```

Figure 7.10. Example of an automatically generated evolvable type: ImageProcCard

[36] In C#, as in other languages (e.g. Java), the instantiation of classes is performed through constructors: special methods which have the same name of the class that they instantiate. The service *New* that is defined in PRISMA specifications and that is provided by the Builder aspect, is implemented as a constructor.

However, note that this class does not contain any specific behaviour of the *ImageProcCard* type (such as internal composition, the ports published, its attributes, etc.). Instead, it only contain invocations to services provided by a *meta* object: *BuildInstance* and *DestroyInstance*. This *meta* object is the meta-instance of the *ImageProcCard* type (i.e. a *IMetaComponent* object), and it encapsulates the specific behaviour of the *ImageProcCard* type, inside the *Builder* aspect.

The *Builder* aspect is implemented as an automatically generated class, called *ImageProcCardBuilder* (see Figure 7.11). This class defines the specific behaviour of the *ImageProcCard* type, and is regenerated each time the type changes. For this reason, and to avoid name conflicts among versions, version number is appended to the name of the class: e.g. *ImageProcCardBuilder_v0* identifies the initial version of the *ImageProcCardBuilder*.

This class contains the following elements:

- Information about the type version that this class implements. It is stored in an attribute called *typeVersion*.
- Information about the type that cannot be automatically obtained by reflection from the generated code, or it is difficult to obtain (e.g. because it has been transformed to platform dependent code). For instance, information about constraints in composite components, the high-level specification of the constructor, etc. This information is defined in the constructor of the Builder class (see the method *ImageProcCardBuilder*) and is stored in a *SimpleSpec* data structure (see Figure 7.3 for more information about this data structure). Later, this data structure will be used by the *TypeDescription* aspect to complete the reified information.
- The method *BuildInstance*, which contains the code that builds and initializes an *ImageProcCard* instance in the PRISMANET middleware. This is done by: (i) importing and instantiating each one of the aspects that the type is composed of (see the *AddAspect* operation), (ii) creating the weavings among these aspects (see the *AddWeaving* operation), and (iii) creating the input/output ports (see the *AddPort* operation).
- The method *DestroyInstance*, which contains the code that destroys an instance, as defined in the PRISMA specification.

```
public class ImageProcCardBuilder_v0 : IBuilder {
   private int typeVersion = 0;
   public int GetVersion { get {return typeVersion;}}

   SimpleSpec specification;
   public SimpleSpec Specification { get {return specification;}}

   public ImageProcCardBuilder_v0() {
      // Definition of relevant type information
      this.specification = new SimpleSpec(typeof(ImageProcCard),
         "AgroBot", AElementType.Component);
      specification.defineConstructor(
         new ParamInfo[] {
            new ParamInfo("cameraPosition", typeof(string))
         },
         new ExpressionSpec[] {
            new ExpressionSpec(
               "ImageProcCardController.begin(cameraPosition);"),
            new ExpressionSpec("ImageProcCardGUI.begin();")
         }
      )
      specification.defineDestructor(
         new ExpressionSpec[] {
            new ExpressionSpec("ImageProcCardGUI.end();"),
            new ExpressionSpec("ImageProcCardController.end();")
         }
      )
   }

   public void BuildInstance(ImageProcCard comp,
                     string cameraPosition)
   {
      // Building and initialisation of instances
      IAspect aspect1= new ImageProcCardController(cameraPosition);
      comp.AddAspect(aspect1);

      IAspect aspect2= new ImageProcCardGUI();
      comp.AddAspect(aspect2);

      comp.AddWeaving(aspect2, "showImage", "image",
         WeavingType.AFTER, aspect1, "newProcessedImage", "image");

      comp.AddPort("VideoIn", "I_VideoServices", "VIDEOCARD");
      comp.AddPort("ImageOut", "I_ImageProcessingServices",
         "IMAGEANALYZER");
   }

   public void DestroyInstance(ImageProcCard comp) {
      comp.GetAspect(typeOf(ImageProcCardGUI)).Dispose();
      comp.GetAspect(typeOf(ImageProcCardController)).Dispose();
      comp.Dispose();
   }
}
```

Figure 7.11. Example of an automatically generated Builder aspect: ImageProcCardBuilder

The reason why two classes are generated instead of only one is that the former, *ImageProcCard*, implements the immutable part of the type (i.e. its name and the kind of architectural element –simple or composite), whereas the latter, *ImageProcCardBuilder*, implements the parts of the type that can be evolved over time (i.e. the type specification).

Type Description Aspect

The *Type Description* aspect provides a reification of the current type version, allowing its inspection and edition at runtime. This reification is kept through several data structures, which capture all the relevant information about the type specification. These data structures are defined as attributes in the *Type Description* aspect, which are populated through the execution of the *Reify* service. The *Reify* service does not only populates these data structures, but also returns a *<Type>Spec* object that allows us to safely edit the type reification. In addition, the *TypeDescription* aspect provides services to inspect only certain elements of a reification, in case an edition is not needed. This is provided through introspection services. Finally, this aspect also maintains the old type versions, allowing non-updated instances to continue executing.

Next, these features are described in detail.

♦ Reification structure

The information that is reified from a type varies depending on the kind of architectural element (see appendix A.3.3 for more details). For instance, in case of *simple* PRISMA architectural elements (i.e. Components and Connectors), the reification is described through the following attributes (see Figure 7.12):

- *Aspects*: a list with the aspect types that the component or connector imports. The information of each aspect is stored in a data structure called *AspectInfo*: i.e. its name, a reference to the type definition, the concern, the interfaces implemented and the played roles provided.
- *Weavings*: a list with the weavings among aspect types. For each weaving, a data structure called *WeavingInfo* stores the name of the aspects that are synchronised, the methods that are weaved, its parameters, and the weaving type (i.e. before, after, instead, etc.).
- *Ports*: a list with the ports that provide or require services from other architectural types. For each port, a data structure called *PortInfo* stores the name of the port, its interface, and its played_role.
- *Constructors*: a list which the definition of the different constructors of the type (i.e. the *new* services defined in the type specification). For

each constructor, the data structure *ConstructorInfo* keeps two lists: one for the initialisation parameters and another for the different expressions defined in the specifications (encoded as strings).

- *Destructor*: a list with data structures *DestructorInfo*, which store the expressions that perform the destruction of the type (i.e. the *destroy* service defined in the type specification).

The information kept for each kind of element is further detailed in appendix A.3.1.1 (see page 418).

```
TypeDescription Aspect SimpleTypeDescription
   using I_SimpleTypeDescription

   Attributes
      Constant
         typeName: string;
         kind: string; // Component or Connector

      Variable
         // Data structures for type reification
         currentVersion: int;
         aspects: list(AspectInfo);
         weavings: list(WeavingInfo);
         ports: list(PortInfo);
         constructors: list(ConstructorInfo);
         destructor: list(DestructorInfo);

         // Auxiliary attributes
         currentTypeVersionFile: string;
            // File with the binary code of the current version
         typeSpecObject: SimpleSpec;
            // Keeps the last generated version of SimpleSpec
         reificationIsValid: boolean;
            // True if data stored about the type is still valid
         reificationsBlocked: boolean;
            // If true, reify operations are temporarily suspended

         // Version management
         oldVersions: list(TypeVersion);

   Services
      in begin(typeName: string,kind: string)
         Valuations
            [begin(typeName,kind]
            this.typeName=typeName;
            this.kind=kind;

      in Reify(output reification: SimpleSpec)
         // Ommitted...

      in Stop()        // Blocks reifications temporarily
         Valuations
            [Stop()] reificationsBlocked=true;

      in VersionChanged(newTypeDefinition:string,
               versionDiffs: list(STEvolutionStep))
         Valuations
            [VersionChanged(newTypeDefinition,versionDiffs)]
            currentTypeVersionFile=newTypeDefinition;
            oldVersions.add(new TypeVersion(currentVersion+1,
               newTypeDefinition, versionDiffs);
            reificationIsValid=false;
            reificationsBlocked=false;

      in GetIncrementalChanges(sourceVersion: int,
               targetVersion: int,
               output versionDiffs: list(STEvolutionStep));
      ...
...
End TypeDescription Aspect SimpleTypeDescription;
```

Figure 7.12. Fragment of the TypeDescription aspect of simple arch. elements

♦ The Reify service

A reification of the type (and the initialisation of the previous attributes) is obtained when the service *Reify* is invoked (see its signature in Figure 7.12). This service performs the following actions:

1) Inspects the executable code of the current type version (implemented in the Builder aspect), gathering the relevant data (i.e. aspect names and types, ports, weavings, etc.). The name of the file that contains the executable code of the current type version is kept in the attribute *currentTypeVersionFile*. When a new type version is generated, this attribute is also updated.

2) Updates the attributes of the aspect with the collected data.

3) Builds a <*Type*>*Spec* object (i.e. a SimpleSpec object in case of simple architectural elements) that encapsulates the reification and allows its manipulation. It is stored in the attribute *typeSpecObject*.

4) Returns the *typeSpecObject* object to the user.

Since the type remains the same until it is evolved, and the reification operation is time consuming (due to the code analysis step), this process has been optimized: the calculations (steps 1 to 3) are only performed one time among evolutions. That is, the first time the *Reify* service is executed, it performs all the calculations (i.e. code inspection, attribute updating and <*Type*>*Spec* building). However, in the following executions, the *Reify* service will only perform the 4th operation: it will return a copy of the <*Type*>*Spec* object generated in the previous call and that was stored in the attribute *typeSpecObject*. Only when the type is evolved and a new type version generated, the *Reify* service would perform again the reification calculations.

This is controlled through an attribute called *reificationIsValid*, which is set to false when the service *VersionChanged* is called. This service is called when a new type version has been generated, and makes the *TypeDescription* aspect to update its reification.

♦ Introspection services

In addition, the *TypeDescription* aspect provides a set of services to allow instances to introspect themselves (see Figure 7.13). These services return information about a specific element of the type, without facilities to edit them: *getAspectTypes*, *getAspectTypeProperties*, *getPorts*, etc. These services are defined in the interface *I_SimpleTypeDescription* (see appendix A.3.2.1), and are published through a port called *TypeIntrospectionPort*, which is only available to instances of the type.

```
TypeDescription Aspect SimpleTypeDescription
   using I_SimpleTypeDescription
...
   Services
      // Services offered through the TypeIntrospectionPort
      in getPorts(output portsList: list);
      in getAspectTypes(output aspectTypesList: list);
      in getWeavings(output weavingList: list);

      in getAspectTypeProperties(aspectName: string,
            output typeDefinition: string, output concern: string,
            output interfaces: list, output playedRoles: list);
      in getPortProperties(portName: string,
            output isProvided: boolean, output isRequired: boolean,
            output interface: string, output playedRole: string);
      in getWeavingProperties(id: string,
         output sourceAspect: string, output sourceMethod: string,
         output sourceParameters: list, output weavingType: string,
         output targetAspect: string, output targetMethod: string,
         output targetParameters: list, output Functions: list);
...
End TypeDescription Aspect SimpleTypeDescription;
```

Figure 7.13. Introspection services provided by a SimpleTypeDescription aspect

♦ Management of type versions

Each time the type is evolved, a new type version (i.e. a new specification) is generated. The *TypeDescription* aspect keeps track of old type versions, to support type conformance of old instances until they are transformed/migrated to the new type version.

Old type versions are stored in the list *oldVersions*. For each type version, the following information is stored (see the *TypeVersion* data structure in appendix A.3.7.1):

- *versionID*: Identifies a type version among others.
- *typeDefinition*: Name of the file that contains the executable code of the type version.
- *versionDiffs*: Set of evolution steps to transform instances from a previous version to this version.

Type versions are identified from each other by means of a version number, which is increased each time an evolution process is applied on the type successfully. The version number is a natural number, where the number zero identifies the first type version. The decision for using natural, consecutive numbers to identify type versions has been motivated to facilitate the ordering of type versions. Since evolutionary changes are sent asynchronously to instances, the version number helps each instance to identify when a previous evolutionary change has not been received correctly. For instance, if an instance is conformant to type version 4, and it is asked to evolve to version 6, it can be easily deduced that one evolution request has not been correctly received. Thus, the instance should ask the type meta-instance to be provided first with the set of evolutionary changes to evolve to version 5, and after that, it could evolve to version 6.

The management of old type versions is required not only to prevent the possible loss of evolution requests, but also to enable the reentering of these instances that had been excluded from evolution. For example, an instance could have been excluded from evolution at version 4 and later be required to evolve to version 7. Since evolutions are propagated to the instance-level as differences among versions, in case an instance has been excluded from evolution, then to integrate it again the instance should be provided with all the intermediate evolution steps. In the previous example, this means that the instance should be provided with the changes that have been introduced in versions 5 and 6. For these cases, the service *GetIncrementalChanges* is provided. Given the IDs of two versions (where the second is subsequent of the first), this service returns the set of evolutionary steps that have been performed among these two versions.

Next, the process of how a type is evolved at runtime is addressed.

Type Evolution Aspect

As described in section 7.3.2.4 (Evolution Process Overview), to evolve a type three steps must be followed by an evolution agent:

1) Obtain a reification of the type to evolve (i.e. a *<Type>Spec* object), which is provided by the *Type Description* aspect;
2) Manipulate this reification, which is performed through the services that the *<Type>Spec* object provides (see these services in appendix A.3.1); and
3) Reflect the changes to the meta-level, which is performed by returning the *<Type>Spec* object to the type meta-instance.

The last step initiates an evolution process, which in turn consists of the following steps (which are automatically performed by the evolution infrastructure):

1) Receive and process the set of evolutionary changes,
2) Generate a new type version,
3) Propagate the changes to the instance-level, and
4) Evolve each instance.

These steps are distributed among different elements of the evolution infrastructure. The *TypeEvolution* aspect is which performs the two first steps: it receives and processes the set of evolutionary changes, and generates a new type version.

♦ The Reflect Service: Starting an Evolution Process

An evolution process is started when the *Reflect* service, published by the type meta-instance and implemented by this aspect, is executed. This service (see its signature in Figure 7.14) requires two input parameters: a *<Type>Spec* object, which provides an edited reification of the type, and an *EvolutionPolicy*, which defines how the evolution process should be executed.

On the one hand, the *<Type>Spec* object is required because it provides the input of the evolution process: it encapsulates a set of attributes (*aspectList*, *weavingList*, *portList*, *constructorList*, and *destructor*, see appendix A.3.1) that describe the new type version that must be generated. On the other hand, the data structure *EvolutionPolicy* allows the evolution agent to adjust some parameters of the evolution process, such as the selection of the strategy for evolving instances or the definition of time constraints for the evolution of instances. Since these parameters deal with the instance-level evolution, which is supervised by the *Evolution Monitoring* aspect, these will be described later in page 293.

Next, the execution of the *Reflect* service is described.

```
TypeEvolution Aspect SimpleTypeEvolution

   Attributes
      Variable
         // Auxiliary variables
         isEvolving: boolean; // A type version is being generated

         // Code generation templates
         builderGenerationTemplate: string;

   Services
      in begin(codeTemplates: string)
         Valuations
            [begin(codeTemplates]
            builderGenerationTemplate=codeTemplates;
            isEvolving=false;

      in Reflect(specification: SimpleSpec,
                     evolParams: EvolutionPolicy);
         Valuations
            // Ommitted...

      out NewVersionGenerated(versionID: int, typeDefFile: string,
            versionDiffs: list(STEvolutionStep),
            evolPolicy: EvolutionPolicy);
...
End TypeEvolution Aspect SimpleTypeEvolution;
```

Figure 7.14. Fragment of the TypeEvolution aspect of simple arch. elements

♦ The Reflect Service: Generation of a new Type version

The *Reflect* service performs several actions: (i) the blocking of other concerns (e.g. the *Builder*, to avoid the creation of new instances during evolution), (ii) the evaluation of preconditions, to guarantee that the evolution process can be executed correctly, (iii) the generation of a new type version, (iv) the dynamic instantiation of the new type version inside the type meta-instance, and (v) the unblocking of other concerns and the propagation of changes to the instance-level.

Next, these actions are described in detail.

Step 1: Blocking of related concerns

During the generation of a new type version, two functional concerns of the type meta-instance are going to be changed: the executable code for creating instances (which is encapsulated in the *Builder* aspect), and the type description (which is offered by the *TypeDescription* aspect). These concerns must be blocked temporarily (i.e. until the generation of the new type version), to avoid that new instances and/or type reifications of the version that is going to be evolved may be created or obtained, respectively.

This blocking is performed through two weavings among the *Type Evolution* aspect and the aspects *Builder* and *Type Description* (see Figure 7.15). The latter aspects must be stopped **before** the *Reflect* service is executed. This stopping means that all the new requests involving the creation of instances or the reification of a type will be buffered and its execution postponed[37]. Thus, once these aspects are stopped, the *Reflect* service can be safely executed. This is captured by the first two weavings (see Figure 7.15).

```
Weavings
   ...
   TypeDescription.Stop() before TypeEvolution.Reflect(spec,params);

   Builder.Stop() before TypeEvolution.Reflect(spec,params);

   // ... more weavings
End_Weavings;
```

Figure 7.15. Weavings used to temporarily block other meta-instance concerns

Step 2: Evaluation of evolution preconditions

To start an evolution process, the reification encapsulated in the *<Type>Spec* object and provided to the *Reflect* service must satisfy the following conditions:

1) The reification must represent the type that this meta-instance manages. This is to avoid the generation of totally different types; only incremental evolutions are allowed. This is evaluated by comparing the attribute *typeName* of the *<Type>Spec* object with the same attribute provided by the *TypeDescription* aspect.

   For instance, the following operation would violate this precondition:

```
         VisionWatchdog.Reify(out reification);
         ImageProcCard.Reflect(reification, ...);
```

2) The type version that the reification modifies is the last type version. That is, the type meta-instance has not evolved yet the version that the evolver agent is trying to evolve. This guarantees that only a single evolution path can be performed: in presence of multiple evolution agents, only the first one in reflecting changes initiates an evolution

[37] This is possible because all PRISMA service invocations are performed asynchronously. See the implementation of the PRISMA model in (Costa-Soria, 2005)

process. The others, when trying to reflect their changes, will be notified about the use of an outdated type version and asked for obtaining a new reification (see section "Single Evolution Path", in page 271). This is evaluated by comparing the attribute *version* of the *<Type>Spec* object with the same attribute provided by the *TypeDescription* aspect.

For instance, in the following code, the evolution agent *evolAgentB* could not reflect its changes:

```
evolAgentA:  ImageProcCard.Reify(out reification1);
evolAgentB:  ImageProcCard.Reify(out reification2);
evolAgentA:  reification1.addAspect(aspA, params);
evolAgentA:  ImageProcCard.Reflect(reification1, ...);
evolAgentB:  reification2.addAspect(aspB, params);
~~evolAgentB:  ImageProcCard.Reflect(reification2, ...);~~
```

3) A new evolution process cannot be started whereas another is generating a type version. Any subsequent *Reflect* request received while another *Reflect* request is being executed, is postponed. If the executing *Reflect* request fails, the first request postponed is then executed. Otherwise, all the postponed requests are cancelled: they edit a type version that is now outdated, so they should be revised and adapted to the new type version.

4) The reification must contain one change at least. Otherwise, it makes no sense to start an evolution process. This is evaluated by checking the number of elements in the list *evolutionSteps* of the *<Type>Spec* object. This list contains the set of evolution steps performed on the original reification.

   For instance, the following operation would violate this precondition:

```
    ImageProcCard.Reify(out reification);
    ImageProcCard.Reflect(reification, ...);
```

If any of these conditions is not fulfilled, the evolution process is aborted and an exception is sent to the evolution agent. If all the evolution preconditions are fulfilled, the next step is the dynamic generation of the new type version, according to the reification provided in the *<Type>Spec* object.

Step 3: Generation of the new type version

As described in previous sections, the executable code of the current type version is encapsulated in the *Builder* aspect. Therefore, to change this type version and introduce a new, evolved type version, the *Builder* aspect must be replaced. The *TypeEvolution* aspect carries out the dynamic generation of the

Builder aspect, which includes the new changes, and its replacement at runtime.

This can be realized because *the TypeEvolution aspect encapsulates the knowledge of how to transform technology-independent concepts* (i.e. the reifications, described in PRISMA ADL) *to technology-dependent concepts* (i.e. the executable code in C#). This knowledge is materialised in a set of **code generation patterns** which regenerate the type when it is evolved at runtime. These patterns define: (i) the code that will not change among versions (e.g. the structure and internal methods of the *Builder* aspect), and (ii) a set of placeholders or tags (encoded with the symbol "%") which will be replaced by code generated from the edited reification. Code-generation patterns are stored in an attribute of the *TypeEvolution* aspect called *builderGenerationTemplate*. These code-generation patterns are generic for all PRISMA elements.

For instance, Figure 7.16 shows the code generation pattern that generates the *Builder* aspect of simple architectural types in C# (see (Hervás-Muñoz, 2009) for further details).

```
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Components.Weavings;
using PRISMA.Middleware;

[assembly:AssemblyVersionAttribute("%__Version__%")]
namespace %__namespace__% {

public class %__TypeName__%Builder_v%__Version__% : IBuilder {
   private int typeVersion = %__Version__%;
   public int GetVersion { get {return typeVersion;}}

   SimpleSpec specification;
   public SimpleSpec Specification { get {return specification;}}

   public %__TypeName__%Builder_v%__Version__%() {
      // Definition of relevant type information
      this.specification = new SimpleSpec(
         (typeof(%__TypeName__%), "%__namespace__%",
         %__AEType__%);
      // Addition of meta-information to the SimpleSpec object
      %__Specification__%
   }

   public void BuildInstance(IComponent comp,
         %__ConstructorParametersDefinition__%)
   {
      // Building and initialisation of instances
      %__ComponentCreation__%
   }

   public void DestroyInstance(%__TypeName__% comp) {
      %__ComponentDestruction__%
      comp.Dispose();
   }
}}
```

Figure 7.16. Code generation pattern for Builder aspects of simple types

For instance, Figure 7.17 shows a fragment of C# code that produces the code related to aspects. In this fragment, *specification* is a variable that contains the reification provided by the evolution agent, and from which the list of aspects (*AspectList*) is obtained. For each aspect, this code obtains the type of the aspect (*AspectType*) and the parameters required for initializing this aspect (*Parameters*). The code is produced as a string and stored in the variable called *componentCreation*, which later (when all the elements of the reification have been processed) replaces the placeholder *%__ComponentCreation__%*.

```
StringBuilder componentCreationAspect;
for (int i=0; i<this.specification.AspectList.Count; i++) {
   AspectSpec aux = (AspectSpec) this.specification.AspectList[i];
   componentCreationAspect = new StringBuilder(
      "comp.AddAspect(new %__AspectType__%(%__Parameters__%));\n");
   componentCreationAspect.Replace("%__AspectType__%",
      aux.AspectType.ToString());
   componentCreationAspect.Replace("%__Parameters__%",
      aux.Parameters);
   componentCreation.Append(componentCreationAspect);
}
```

Figure 7.17. Fragment of the code generation process for aspects in C#

It is important to note that the generated code must also include *meta information* about the type, to later allow the complete reification of the original type specification. Meta information is this information contained in a high-abstraction level specification that is not directly transformed to code, or which could be very difficult to obtain from the executable code. An example of this meta-information is the high-level specification of behaviour, such as the definition of constructors in architectural elements (i.e. the *New* service), or the definition of services in aspect types. Another example of this meta-information is the specification of the cardinality constraints that are defined in composite components. In our implementation, this meta-information is provided statically in the constructor of the *Builder* aspect and stored in a *<Type>Spec* object (see the placeholder called "*%__Specification__%*"

in Figure 7.16). This object is dynamically obtained by the *Reify* service when it inspects the *Builder* aspect.

Step 4: Dynamic instantiation of the new code

Once the source code of the new *Builder* aspect has been generated, the next step is the dynamic compiling and linking of this code to the running meta-instance.

On the one hand, the dynamic compilation of the new code, in the PRISMANET implementation (see (Hervás-Muñoz, 2009), has been performed by means of the CodeDom[38] library. This library allows to dynamically invoke the C# compiler to produce a new assembly (a file that contains executable code). The new assembly produced is stored in the file system together the rest of assemblies of the type, which makes the changes performed to the *Builder* aspect permanent. In case the system is shutdown and restarted again (or the type is transferred over the network), the loading of the type will always contain the latest evolutions, because this loading will always use the latest *Builder* aspect.

On the other hand, the dynamic linking and instantiation of the new code into the running type meta-instance is performed through the services that the PRISMANET middleware provides to dynamically modify running PRISMA instances. For instance, the service *ReplaceAspect* allows us to replace an aspect at runtime while keeping its existing weavings, whereas the service *GetPRISMAType* allows us to dynamically load the last version of the type which name is provided:

```
    meta.ReplaceAspect("ImageProcCardBuilder",
                           GetPRISMAType("ImageProcCardBuilder"));
```

The replacement can be safely performed, since the old *Builder* aspect has been previously stopped. In this way, the old *Builder* aspect is replaced at runtime, and then the executable code of the previous type version.

Step 5: Unblocking of related concerns

At this point, the new type version has been generated and weaved to the running meta-instance. The final step is the unblocking of these processes that had been previously blocked (i.e. the creation of instances and the reification of the type), and the propagation of changes to the instance-level. Since these

[38]http://msdn.microsoft.com/en-us/library/system.codedom.aspx; http://msdn.microsoft.com/en-us/library/650ax5cx.aspx

are operations that are related to other concerns of the meta-instance (i.e. the aspects *Builder*, *Type Description*, and *Evolution Monitoring,* respectively), this step is not directly performed by the *Reflect* service, but by weavings.

When a new type version is generated, the *Reflect* service calls the service *NewVersionGenerated*, providing the following information (see its signature in Figure 7.14, page 286):

- *versionID*, *typeDefFile*: The ID of the new type version and the name of the assembly that has been generated.
- *versionDiffs*: The set of differences respect to the previous type version. These differences will be propagated to the instance-level to allow each instance to integrate only the incremental changes.
- *evolPolicy*: The evolution policy that has been provided when calling the *Reflect* service. This will be used by the *Evolution Monitoring* aspect.

When this service is called, a set of weavings are activated (see Figure 7.18), which in turn initiate other services:

- The service *Start* of the *Builder* aspect. This enables the creation of new instances, according to the new type versions. The instantiation requests that had been postponed are unblocked and executed.

```
Weavings
   ...
   Builder.Start()
      after
   TypeEvolution.NewVersionGenerated(version,type,diffs,evPolicy);

   TypeDescription.VersionChanged(type, diffs)
      after
   TypeEvolution.NewVersionGenerated(version,type,diffs,evPolicy);

   EvolutionMonitoring.ReflectToInstances(version, diffs, evPolicy)
      after
   TypeEvolution.NewVersionGenerated(version,type,diffs,evPolicy);

   // ... more weavings
End_Weavings;
```

Figure 7.18. Weavings activated when the Reflect service finishes

- The service *VersionChanged* of the *Type Description* aspect. This services stores the details of the new type version, allowing its later reification and version management. This service also unblocks the reification requests, which will return the reification of the new type version, not the old one.

- The service *ReflectToInstances* of the *Evolution Monitoring* aspect. This service initiates the propagation of changes to the instance-level.

From this point in time, although the instances may have not integrated the new changes, the evolution of the type is finished: a new type version has been generated, new instantiations of this version can be created, and old instances have been notified to start a migration process to integrate the new changes. Then, the *Type Evolution* aspect is available to accept new evolution requests.

Evolution Monitoring Aspect

The *Evolution Monitoring* aspect keeps the asynchronous connection with the instance-level: it manages the population of instances, propagates evolutions to these instances, and supervises periodically the evolution of these instances. Next these features are described.

```
EvolutionMonitoring Aspect SimpleInstancesMonitoring

   Attributes
      Variable
         // Instance population
         population: list(InstanceInfo);
         // List of time-constrained active evolution processes
         activeEvolutionProcesses: list(EvolutionProcess);

   Services
      in ReflectToInstances(version: int,
            evolutionSteps: list(STEvolutionStep),
            evolutionParams: EvolutionPolicy);
         Valuations
            // Ommitted...

      EvolutionProcessMonitoring(evProcess: EvolutionProcess);
         // This private service is executed periodically to
         // supervise the evolution of instances
      suspend(timeout : integer);
         // This function suspends the current process

      in RegisterInstance(instanceRef: InstanceInfo)
         Valuations
         [RegisterInstance(instanceRef)]
         population.add(instanceRef);

      in UnRegisterInstance(instanceRef: InstanceInfo)
         Valuations
         [UnRegisterInstance(instanceRef)]
         population.remove(instanceRef);
      ...
...
End EvolutionMonitoring Aspect SimpleInstancesMonitoring;
```

Figure 7.19. Fragment of the EvolutionMonitoring aspect of simple arch. Elements

♦ Population Management

This aspect keeps the connection of the type-level with the instance-level: it manages the references to the type instantiations that have been created and are still running. Whenever a new instance is created, the service *RegisterInstance* is executed, and a reference to the new instance is stored in the attribute called *population* (see Figure 7.19, *Variable* and *Services* section). Likewise, if an instance is no longer needed and is destroyed, the service *UnRegisterInstance* is executed, so the reference to this instance is removed from the population. The triggering of the services *RegisterInstance* and *UnRegisterInstance* is performed by the weavings defined in the meta-instance among the aspects *Builder* and *EvolutionMonitoring* (see Figure 7.9, page 275).

♦ Propagation of evolutions to the Instance-Level

This aspect provides a service called *ReflectToInstances*, which is executed when a new type version has been generated. This service is provided with three parameters:

- *version*, the code of the new type version, so instances can identify and correctly order multiple evolution requests.
- *evolutionSteps*, which define the set of evolution steps to apply on each instance, and
- *evolutionParams*, the evolution policies that will guide the evolution of instances.

Next the two last parameters are described in detail, because they encode how the evolution of instances should be realized.

Definition of evolution steps

On the one hand, the parameter *evolutionSteps* provides a list containing the incremental changes to perform on the structure of each instance. Each incremental change is stored in a data structure called *STEvolutionStep* (for changes in simple architectural types; for composite types this data structure is called *CTEvolutionStep*). This data structure contains the following attributes (see Figure 7.20):

- *Action*, the evolution operation to perform on the instance: an addition (*Add*), a removal (*Remove*), an update (*Replace*), the starting of an element (*Start*) or the stopping (*Stop*).
- *Type*, the kind of structural element of an instance on which the action must be performed: an aspect, a port, or a weaving.

- *Data*, a specification with more information about the element of type *Type*, on which *Action* must be performed. For instance, if the subject of an evolution operation is an aspect, this attribute will contain an *AspectInfo* data structure, with the name of the aspect to add or remove, its parameters, etc. In case of replacement, the data structure *ReplacementInfo* contains the information of the old aspect to replace, and the new aspect.

```
Data structure STEvolutionStep
   Attributes
      action : {Add | Remove | Replace | Start | Stop};
      type: { Aspect | Port | Weaving };
      data: { AspectInfo | PortInfo | WeavingInfo |
              ReplacementInfo };
End_Data_Structure STEvolutionStep;
```

Figure 7.20. Data structure STEvolutionStep

The service *ReflectToInstances* propagates these evolution steps (i.e. the collection of *STEvolutionStep*) to each instance that must be evolved. To do so, each instance provides a service called *reflectToInstance* (see section "*Instance Evolution Planning Aspect*", page 301), which initiates the transformation of its internal structure (asynchronously with respect to the evolution of the type and the other instances).

Definition of evolution policies

On the other hand, the parameter *evolParams* provides a data structure, *EvolutionPolicy* (see Figure 7.21), which defines the policies that will guide the evolution of instances. These policies define the set of instances to be evolved, and the evolution timeout per instance.

Selection of the instances to evolve. To define the set of instances to evolve, the attribute *evolutionStrategy* is used. This attribute can be set to one of the following values:

- **EvolveAll**. All the existing instances will evolve to the new type version; new instances will use the new version.
- **ExcludeSome**. A subset of instances is exempt from evolving to the new type version; the rest will evolve to the new version. The subset of instances to exempt is defined in the attribute *exclusionSet*.
- **OnlyNew**. Only new instances will use the latest type version. The existing instances will not evolve to the new version.

By default the *EvolveAll* strategy is used (see the initialization section of the *EvolutionPolicy* data structure): each instance of the type is requested to evolve to the latest type version. However, in certain cases it may be need to exclude some instances from evolution. For instance, this is needed when there are compatibility issues of the new version with other existing types. In this case, the strategy *ExcludeSome* should be used. Note that these instances that do not evolve to a version *n*, later can be required to evolve to a subsequent version *m* > *n*. In this case, these instances will apply the sequence of evolutions performed from the version *n* to *m*. This is planned at the instance-level by the *InstanceEvolutionPlanning* aspect (which is described in page 301).

```
Data structure EvolutionPolicy
   Attributes
      // Strategy for evolving instances
      evolutionStrategy : { OnlyNew | EvolveAll | ExcludeSome };
      exclusionSet : list(string);

      // Maximum time each instance has to apply the changes
      evolutionTimeoutPerInstance : int; // in milliseconds
      evolutionRetriesIfTimeoutExceeded : int;
      actionIfRetriesExceeded: { ForceEvolution | AbortEvolution };

   // Default evolution policy: evolve all instances in 1 minute
   // Otherwise, abort the evolution of the instance
   new() {
      evolutionPolicy = EvolveAll;
      exclusionSet = list[];
      evolutionTimeoutPerInstance = 10000;
      evolutionRetriesIfTimeoutExceeded = 6;
      // Maximum total time to evolve: 10000x6=60000 milliseconds
      actionIfRetriesExceeded = AbortEvolution;
   }
End_Data_Structure EvolutionPolicy;
```

Figure 7.21. EvolutionPolicy data structure

Defining time-constrained evolutions. The other evolution policy that can be adjusted through the *EvolutionPolicy* data structure is the available time that instances will have to evolve, and the action to perform in case an instance does not evolve in the provided timeout. This policy is optional, but prevents that an instance could be indefinitely pending to evolve, which would lead the current evolution process to never be finished.

This is defined through the following attributes:

- *evolutionTimeoutPerInstance*: in milliseconds, this attribute defines the maximum time each instance has to evolve to the next type version. If this timeout is set to zero, no evolution timeout is set, so each instance is not time-constrained to evolve.

- *evolutionRetriesIfTimeoutExceeded*: number of opportunities given to each instance to retry its evolution.
- *actionIfRetriesExceeded*: the corrective measure to apply when the timeout has been exceeded and no retries are available. The action can be:
    - **AbortEvolution**, which cancels the evolution of the instance, leaving it in its current type version. This is the action taken by default.
    - **ForceEvolution**, which forces the instance to evolve to the next type version, possibly losing its current state or compatibility with other instances (depending on the reason that motivated the evolution delay).

Thus, an evolution process can be constrained whether to finish in bounded time or not. In case this time is exceeded without evolving to the next version, the *Evolution Monitoring* aspect will send an event to these instances pending to evolve to either force or abort its evolution (depending on the evolution policy).

♦ Monitoring of Evolution Processes

The management of active evolution processes is carried out through the service *EvolutionProcessMonitoring*. This service is periodically executed for each evolution process, in intervals defined by its evolution timeout[39], supervising the instances that are pending to evolve. Each time this service is executed, it is provided with the information of the evolution process that is being monitored (see the data structure *EvolutionProcess*, in Figure 7.22): the target type version, the instances that are pending to evolve to this version, the remaining retries until a corrective measure is applied, and the details of the evolution policy (such as the timeout available to evolve).

This service first updates the list of instances pending to evolve (list *instancesToEvolve*), removing those that have been evolved to the target type version. This can be evaluated because each instance provides an attribute, *currentVersion*, which defines the type version that the instance is conformant to (see the *InstanceInfo* data structure in appendix A.3.5.1).

[39] In case of evolution processes which are not time-constrained (i.e. those with an evolution timeout=0), no monitoring is needed. Instances are free to evolve to the next version or not, since they do not have a maximum time to apply changes. For this reason, the list of active evolution processes only contains those that should be monitored.

```
Data structure EvolutionProcess
   Attributes
      targetVersion: int;
      instancesToEvolve: list(InstanceInfo);
      remainingRetries: int;
      evolutionPolicy: EvolutionPolicy;
End_Data_Structure PendingEvolution;
```

Figure 7.22. EvolutionProcess data structure

Next, if the remaining retries is greater than zero, then this service decreases the remaining retries and programmes itself to execute again in the timeout defined in the evolution process (which is stored in the attribute *evolutionPolicy* of the *EvolutionProcess* data structure).

Otherwise (*remainingRetries*==0), the evolution process must be finished. Then the action defined in the evolution policy is applied on those instances that are still pending to evolve. For each instance of the list *instancesToEvolve*, an event *AbortEvolution* or *ForceEvolution* is sent, waiting until they acknowledge the event. Thus, each instance would either abort or force immediately its evolution. Finally, the information relative to this evolution process is removed from the list *activeEvolutionProcesses* maintained by the *EvolutionMonitoring* aspect.

Note that, since some evolution processes may be time constrained whereas others not, it may happen that a newer evolution process must finish before the previous one has finished. This applies in such a case where an older evolution process has a timeout longer than the newer one (e.g. the older evolution process is not time-constrained). What happens then to these instances that are still evolving to a previous version (i.e. subject to the older evolution process) when the time of the newer evolution process finishes? An approach could be to force the evolution of such instances to the newer version, but this may lead to inconsistencies if instance evolution has been delayed due to type incompatibilities with the new version. And to abort the evolution of such instances is not a reasonable option: when such instances finish the previous evolution process, then they should be given with an opportunity to evolve to the new type version.

The solution to this issue is the following. When the time available for an evolution process expires, the events *abortEvolution* or *forceEvolution* are sent to all the instances that are pending to evolve. However, these events are only executed by these instances that are not subject to other previous evolution processes. That is, these instances which current version is the one that the expiring evolution process is evolving from (i.e. those instances that *currentVersion==targetVersion*−1). The other instances, i.e. those that are still evolving to a previous version, will receive the event but *they will defer its execution until they reach the type version that this event applies to* (see the method *TransformInstance* on the *Instance Evolution Planning* aspect, page 304). That is, the monitoring of the expiring evolution process will finish (as in the case of non time-constrained evolution processes), but each instance will locally handle the evolution. When a delayed instance reaches the type version managed by a time-constrained evolution process that has expired, it immediately applies the action defined in the evolution policy: to force or abort its evolution.

Note that the concept of evolution process is abstract and spans multiple entities: it starts when a new evolution request is processed by the *TypeEvolution* aspect (i.e. the *Reflect* service); it comprises the generation of a new type version (i.e. a new *Builder* aspect), the propagation to each instance (performed by the *EvolutionMonitoring* aspect), and the evolution of instances (performed locally by each instance); and it only finishes when all the instances that have received the evolution steps have integrated the new changes (or aborted the evolution, according to the evolution policy).

Next, it is described how the evolution process is performed at the instance-level, for each one of the instances.

7.3.5.2 Instance-level Evolution

The evolution performed at the instance-level has two main characteristics: it is performed asynchronously, and through a transformational approach.

On the one hand, instance-level evolutions are asynchronous: each instance evolves at different times with respect to type-level evolutions and with respect to other instances (see section 7.3.4). This enables instance evolutions driven by the local context: if the interacting instances are not compatible with the new type version, the instance will remain unevolved. Thus, these non-evolved instances will neither block the evolution of its type nor the evolution of other instances.

On the other hand, instances evolve through (incremental) transformations: the internal structure of each instance is disassembled and adapted to fit the new structure defined by the new type version (see section 7.3.3). Thus, instances evolve transparently from inside, with their physical boundaries remaining unaltered. This is a less disrupting approach that evolving through instance migrations.

This asynchronous, transformational evolution approach can be only supported if instances are provided with mechanisms to: (i) receive and manage themselves the evolution requests received from the type-level, and (ii) disassemble and transform their internal structure. These mechanisms are provided by the following three functional areas:

1) *EvolutionPlanning*: receives the evolution requests from the meta-instance and coordinates when and how they can be performed safely in the instance.

2) *Monitoring*: provides runtime information about the structural elements an instance is composed of.

3) *Effector*: dissassemblies the structural elements of the instance and performs changes on each element.

Since these functional areas identify different concerns of the instance evolution process, they have also been encapsulated into aspects, called *instance-level evolution aspects*.

The integration of instance-level evolution aspects inside each instance[40] varies depending on the kind of architectural element: simple or composite. On the one hand, in case of simple architectural elements (i.e. *Component* or *Connector* instances), instance-level evolution aspects are called: *InstanceEvolutionPlanning*, *InstanceMonitoring*, and *InstanceEffector*. Since simple instances are internally described by aspects, the instance-level evolution aspects are integrated together the rest of user-defined aspects, but cannot be changed by the architect. In the implementation, these aspects are integrated as part as the executing PRISMA model and thus they cannot be directly seen in the generated code (see Figure 7.23).

On the other hand, in case of composite architectural elements (i.e. System instances), instance-level evolution aspects are called: *ReconfigurationCoordination*, *ArchitectureMonitoring*, and *ArchitectureEffector*. Since composite instances are described by means of configurations of component and connector instances, the instance-level evolution aspects have been encapsulated in a specific component called *Evolver* (see Figure 6.29, page 225). This component, which has been described in detail in Chapter 6, provides autonomic reconfiguration behaviour to a composite instance. Moreover, this component, through the aspect *ReconfigurationCoordination*, also contains the behaviour that propagates type-level changes to a composite instance.

[40] Note that the instance-level evolution aspects will be only integrated in case the type has been defined as evolvable.

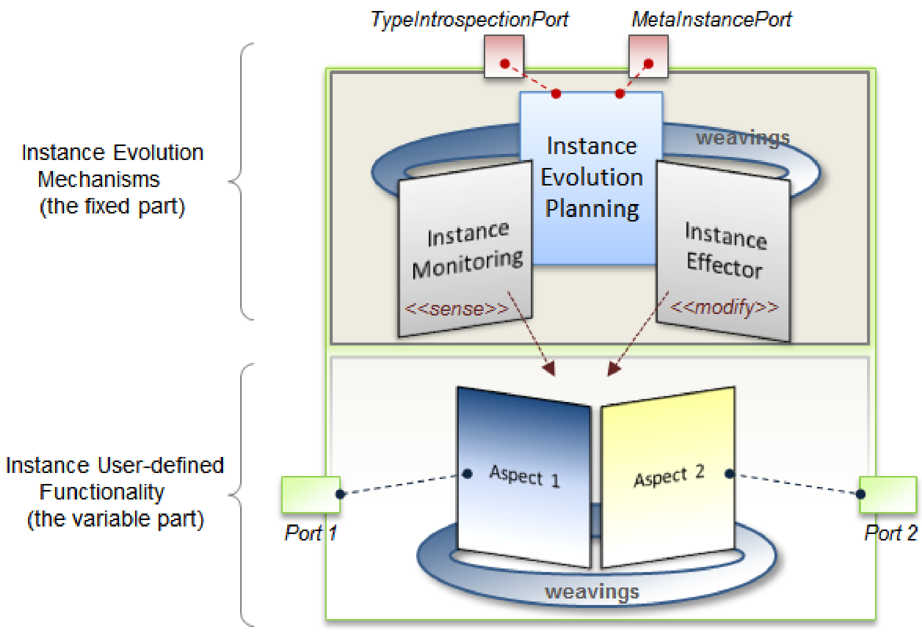


Figure 7.23. Integration of instance-level evolution aspects in simple instances

Since the instance-level evolution aspects of composite instances have been covered in the previous chapter[41], in this chapter only the details of simple instance evolution aspects are covered.

Next, the main characteristics and behaviour of these aspects is described.

Instance Evolution Planning Aspect

The *InstanceEvolutionPlanning* aspect can be considered the delegate of the type-level in each instance: it receives the list of changes that must be applied in the instance, and coordinates how the instance structure must be transformed to integrate these changes safely.

Figure 7.24 shows the main attributes and services that are provided by this aspect, including transaction management, evolution request management, and atomic change operations. A more complete specification is provided in appendix A.3.7 (see page 432).

[41] Although the aspects of the *Evolver* component have not been described in Chapter 6 from the point of view of type-level driven changes, these kinds of changes are also supported. The *ReconfigurationCoordination* aspect provides a service to receive type-level evolution requests, which trigger the regeneration of the domain-specific part, and sends the provided evolutionary steps to the *ArchitectureEffector* aspect, which changes the instance architecture. This shows the versatility of the aspects provided.

```
InstanceEvolutionPlanning Aspect
...

Attributes
   Constant
      instanceID: string;

   Variable
      instance_version: int;  // Current version
      pendingEvolutions: queue(EvolutionRequest);
         // list of pending evolutions: versionID and set of changes
      isEvolving: boolean;
         // If the instance is evolving to another version.

      // Auxiliary variables for evolution process management
      newVersion: int;
      evolutionSteps: list(STEvolutionStep);
      partsToStop: list(string);
      partsToStart: list(string);

      // Variables for transaction management
      ...

Services
   // *** Type-level Interaction Services ***
   in ReflectToInstance(newVersion: int,
         evolutionSteps: list(STEvolutionStep));
   in ForceEvolution(versionID: int);
   in AbortEvolution(versionID: int);

   in GetCurrentVersion(output versionID: int);
   out GetIncrementalChanges(currentVersion: int, targetVersion: int,
         output evolutionSteps: list(STEvolutionStep));

   // *** Internal Evolution Services ***
   TransformInstance();

   // *** Atomic change operations ***
   AddAspect(aspectTypeName: string, parameters: string);
   RemoveAspect(aspectTypeName: string);
   ReplaceAspect(aspectToReplace: string, newAspectType: string,
      newAspectParameters: string);
   AddPort(portName: string, interface: string, playedRole: string);
   RemovePort(portName: string);
   AddWeaving(sourceAspect: string, sourceMethod: string,
      sourceParameters: list, weavingType: string,
      targetAspect: string, targetMethod: string,
      targetParameters: list, transfFunctions: list);
   RemoveWeaving(sourceAspect: string, sourceMethod: string,
      weavingType: string, targetMethod: string);

   // *** Transaction Management ***
   BeginEvolutionTransaction();
   EndEvolutionTransaction();
   RollbackEvolutionTransaction();

   // *** Starting and Stopping services ***
   out StartAspect(aspectID: string);
```

```
    out StopAspect(aspectID: string);
    out StartPort(portName: string);
    out StopPort(portName: string);
    out StopWeaving(sourceAspect: string, targetAspect: string);
    out StartWeaving(sourceAspect: string, targetAspect: string);
...
End_InstanceEvolutionPlanning_Aspect
```

Figure 7.24. Services of the InstanceEvolutionPlanning aspect

Next the most important details about this aspect are described.

♦ ReflectToInstance: Receiving changes from the type-level

Changes are received from the type-level by means of the service *ReflectToInstance*. This service is invoked to notify the instance of the generation of a new type version, so that the instance would evolve to this version as soon as possible. The service *ReflectToInstance* requires two input parameters:

- *newVersion*: an integer that identifies the new type version;
- *evolutionSteps*: the list of incremental changes (i.e. STEvolutionStep elements, see their structure in appendix A.3.1.4) to evolve from the previous type version to the new type version (i.e. *newVersion*)

Since instances evolve asynchronously, the invocation of the *ReflectToInstance* service does not mean that the instance would evolve immediately. By the contrary, this service stores the information related to the evolution request (i.e. the pair *newVersion* and *evolutionSteps*) in a data structure called *EvolutionRequest* (see its specification in appendix A.3.7.1, page 432). This data structure is added to an internal list, called *pendingEvolutions*, for its later processing:

```
in ReflectToInstance(newVersion: int,
                     evolutionSteps: list(STEvolutionStep))
   Valuations
      [ReflectToInstance(newVersion, evolSteps)]
      pendingEvolutions.Add( new EvolutionRequest(
            targetVersion=newVersion, evolutionSteps=evolSteps));
```

One of the reasons to do that is to acknowledge the caller (i.e. the type meta-instance) as soon as possible that the evolution request has been received successfully. The other reason is to guarantee that no concurrent instance evolutions are performed: a different thread is in charge of periodically processing the evolution requests received.

♦ TransformInstance: The execution of an evolution process

When a new evolution request is added to the list *pendingEvolutions*, the service *TransformInstance* is executed automatically. This service initiates the evolution of the instance to integrate the changes introduced by a new type version. This evolution means that the instance *transforms its internal structure to the structure defined in the new type version*.

To guarantee that only a single evolution process is being executed, the attribute *isEvolving* is used. This attribute is set to true when an evolution process is initiated, and is set to false when an evolution process finishes. The service *TransformInstance* is only executed when the attribute *isEvolving* is false.

The fragment below shows the trigger that initiates the execution of the *TransformInstance* service (see Figure 7.24, triggers section):

```
// Initiation of an evolution process
TransformInstance() when
   pendingEvolutions.Size()>0 && isEvolving=false;
```

Next, the different steps that are performed by the *TransformInstance* service to transform the instance are described below.

Step 1: Obtain evolution request and evaluate version conformance

The service *TransformInstance* begins by obtaining an evolution request from the list of pending evolutions:

```
TransformInstance()
   Valuations
      [TransformInstance()]
      // Processing of a pending evolution request
      evolutionRequest = pendingEvolutions.dequeue();
      newVersion = evolutionRequest.targetVersion;
      evolutionSteps = evolutionRequest.evolutionSteps;
```

An evolution request describes changes as a set of differences (i.e. *evolution steps*) among two consecutive type versions, the *source version* and the *target version*. Therefore, an evolution request consists of three elements: a source version identifier, a target version identifier, and a set of evolution steps. However, note in the specification shown above that only two elements are obtained: *newVersion*, i.e. the target version identifier, and *evolutionSteps*, i.e. the list of differences among the source and target versions. But no information is gathered about the source version identifier. The reason is because, in our implementation, this is not necessary: the source version identifier is calculated from the target version identifier[42], `source_version = newVersion-1`.

The principle behind our evolution approach is the following: if an instance is *conformant to*[43] the source version identifier, then by applying the set of differences or evolution steps on this instance, this results an instance conformant to the target type version. Otherwise (i.e. an instance does not have this conformance to the source version), the set of evolution steps cannot be used: it may result in inconsistent evolutions (e.g. an evolution step may assume the existence of elements that the current instance does not have).

For this reason, the first step when processing an evolution request is to evaluate if the instance to evolve is conformant to the source version of this request. This conformance is evaluated through the attribute *instance_version* (see Figure 7.24, variables section). This attribute keeps track of the type version that an instance is currently conformant to, and is updated each time that the instance is successfully evolved to a new type version. Therefore, if the attribute instance_version equals to the source version of the evolution request (i.e. `instance_version=newVersion-1`), then this means that the instance is conformant, so the evolution steps can be applied on the instance.

Otherwise, if the instance is not conformant to the source version (i.e. `instance_version<newVersion-1`), this means that the instance has been excluded from previous evolutions or that it has lost previous evolution requests. In this case, the *InstanceEvolutionPlanning* aspect must request the set of changes that have taken place from the current instance version to the source type version, and integrate them with the list of evolution steps received. This is performed through the service *GetIncrementalChanges*, which returns the sequence of evolution steps among two type versions:

```
if (instance_version<(newVersion-1)) then
   ( TYPE_INTROSPECTION.GetIncrementalChanges(instance_version,
                     newVersion, output evolutionSteps)     );
```

As a result, the list of evolution steps, called *evolutionSteps*, is extended with the missing evolution steps. In this way, the description of changes to drive

[42] Recall that, as described in the section "Type Description Aspect" (see page 298), version identifiers are natural numbers that are increased each time a new version is generated.

[43] In this context, it is said that *an instance is conformant to a type version* when the structure and behaviour of the instance satisfies the specification of the type version.

the current instance version to the target version is complete. Next, the runtime transformation of the instance can be started.

Step 2: Preparing the instance for transformation

Before starting the transformation of the instance structure, the instance must be driven to a safe status, to avoid the introduction of inconsistencies due to the transformation process.

The safe status is achieved by stopping (or isolating) the parts of the instance that are going to be changed. The set of parts to stop (i.e. an aspect, a port, or a weaving) are obtained by looking in the list *evolutionSteps* for the parts that are going to be removed, replaced or stopped. The parts found are added to a list called *partsToStop*. Then, each element of this list is driven to a quiescent status (see section 3.4), by executing the services *StopAspect*, *StopWeaving* or *StopPort*, respectively. The implementation of these services is provided by the *InstanceEffector* aspect (which is described later in page 311).

Note that in some cases, the parts that are interacting with a part that is going to be changed (i.e. a part that is in the list *partsToStop*) must be also stopped. For instance, if a port or a weaving is going to be removed, then the aspect that receives/sends service requests from/to this port, or that is intercepted by the weaving, must be also stopped to avoid inconsistencies. This is performed in a second step: the algorithm looks in the list *partsToStop* for removal/replacement operations, and for each part that is going to be removed/replaced, it evaluates if the interacting parts should be also stopped. In affirmative case, the interacting parts are added to the list *partsToStop*.

Since the safe stopping of the parts that are subject to evolution may take time, the service *TransformInstance* suspends itself until the stopping finishes (or it is aborted, in case of forced evolutions, see page 308). When the service is awaked, then the instance transformation process continues.

Step 3: Transforming the instance

The transformation –or evolution– of an instance is performed by means of atomic change operations (additions, removals, or updates) on the structural parts that assemble the instance (i.e. aspects, weavings and ports). The result of this transformation is not only structural, but also behavioural, because the elements that define the behaviour of the instance and their interrelations have been changed.

For each evolution step contained in the list *evolutionSteps* (e.g. addition of an aspect, removal of a weaving, etc.), the service *TransformInstance* executes the corresponding atomic change operation (e.g. *addAspect*, *removeWeaving*, etc.).

Since atomic change operations usually involve low-level changes (i.e. modifying memory structures and pointers), they are encapsulated in a different aspect, the *InstanceEffector* aspect. In this way, the *InstanceEvolutionPlanning* aspect is kept as platform independent as possible.

In addition, the *InstanceEvolutionPlanning* aspect is also responsible for guaranteeing that a transformation process does not finishes unexpectedly, leaving the instance partially transformed. An evolution process must be successfully completed or, if anything fails, reverted to the previous version. This is performed through a transactional management of the transformation process. A transformation process is preceded and concluded by the execution of the following operations, respectively: *BeginEvolutionTransaction* and *EndEvolutionTransaction*. These operations delimit the beginning and the end (i.e. the commit) of an instance evolution process. During the execution of atomic change operations, auxiliary information is kept by the *Instance Evolution Planning* aspect for later undoing the changes if an error occurs. If the transaction finishes successfully (i.e. the *EndEvolutionTransaction* is executed), this information is simply removed and the new changes made permanent. Otherwise, if any change operation fails or triggers an exception, a service called *RollbackEvolutionTransaction* is executed, which uses the previously stored auxiliary information to undo the changes. The details of these services are not described here, because they share a lot of similiarities with the transaction management of reconfiguration plans described in section 6.4.3.2 (page 194).

Therefore, the set of evolution steps provided by the evolution request are translated to atomic change operations and executed inside a transactional context, which if successfully completed will be commited or undone in case of any failure.

Step 4: Finishing the transformation process

Once the transformation process finishes (whether successfully or not), the final step is to unblock the parts of the instance that had been quiesced in the second step or that have been added in the previous step. This is done by means of the services *StartAspect*, *StartWeaving* or *StartPort*, which are provided by the *InstanceEffector* aspect.

Finally, the variable *isEvolving* is set to false, thus allowing that another evolution process may be started.

♦ Forcing or aborting an instance evolution process

As described in "Definition of evolution policies" in page 295, an evolution process may be constrained to finish in bounded time. In case this time is exceeded, the type meta-instance may request two corrective measures: to abort the execution of the evolution process, or to force the completion of the evolution process immediately. The InstanceEvolutionPlanning aspect is in charge of applying these corrective measures.

The corrective measures are invoked through the services *abortEvolution* or *forceEvolution*, which require the identifier of the evolution process to abort or force. This identifier relates to the target version of an evolution request. It is required to distinguish the evolution request to abort or force among the others, since several request may be pending at the same time.

On the one hand, the service *abortEvolution* aborts the execution of an evolution process. The action performed depends on whether the evolution process to abort is currently active or not. If the evolution process to abort is currently active (this can be observed if the value of the attribute *newVersion* equals to the target version of the evolution process to abort), then its execution must be rollbacked. This is achieved by means of two consecutive operations: (i) sending an awakening signal to the service *TransformInstance*, and (ii) executing the service *RollbackEvolutionTransaction.* The former ensures that the service TransformInstance is not waiting for the safe stopping of the instance parts, whereas the latter undoes any evolution step applied on the instance. As a result, the current evolution process is aborted and the instance returned to its previous version. By the contrary, if the evolution process is not currently active (i.e. it is in the list of pending evolutions), the action performed is simply the removal of the corresponding evolution request from the list of pending evolutions. In this way, this evolution request will not be processed.

On the other hand, the service *forceEvolution* forces the completion of an evolution process, without taking care of maintaining the current state of the instance or aborting running services. The action performed depends on whether the evolution process to complete is currently active or not. If the evolution process is currently active, then the delaying operations must be minimized, to finish the process as soon as possible. Since the most delaying operation is the safe stopping of the instance, this is aborted, despite of possibly leading to inconsistencies in the instance state or the cancelling of running services. This can be solved by reinitializing the state of the instance to its predefined values. By the contrary, if the evolution process to complete is not currently active (i.e. the corresponding evolution request is in the list of pending evolutions), its completion must be postponed to the moment where it would be executed. In this case, the evolution request is tagged appropriately: the attribute *evolutionFlags* is set to "forceEvolution". Later, when this evolution request is processed, this attribute will trigger the activation of a timer for the safe stopping operation. If the safe stopping is not performed before this timer finishes, then the safe stopping will be aborted and the completion of the transformation process will be forced.

Thus, it can be seen that the *InstanceEvolutionPlanning* aspect plays a key role in the asynchronous evolution process: it interacts with the type-level and acts locally in the context of an instance. In addition, the aspect also provides mechanisms to rollback the evolution in case anything goes wrong: for instance, an evolution process does not finish successfully, or an abort signal is received from the type-level. And all of this is performed without addressing low-level details. These details are provided by two different aspects, the *InstanceMonitoring* aspect and the *InstanceEffector* aspect, which are described below.

Instance Monitoring aspect

The *InstanceMonitoring* aspect provides information about the current instance structure and its execution context. The information gathered depends on each structural part (i.e. aspects, weavings and ports), and concerns their attributes, current execution status, and the dependencies to other structural parts. For instance, Figure 7.25 shows the most relevant services provided by the *InstanceMonitoring* aspect.

Services 1 to 3 provide the list of IDs of each kind of structural part of the instance (aspect, port or weaving). These services allow us to iterate over the different parts an instance is composed of.

Services 4 to 6 provide the reference (i.e. the memory pointer, an object) to a specific structural part, so that it can be manipulated by the evolution services. In order to make easy the analysis of evolution dependencies, these services also provide the reference to the parts that are linked together. For instance, when an aspect is encapsulated in a component and one of its played roles is assigned to a component port, a relationship among the aspect and the port is created, which must be taken into account in the evolution process. For this reason, the service *getAspectProperties* provides the list of ports that are linked to an aspect, through the parameter *linkedPorts*. Similarly, the service *getPortProperties* returns not only the pointer to the required port, but also the aspect ID that the port publishes services from.

```
InstanceMonitoring Aspect
   ...
   Services
1     in getAspects(output aspectIDs: list);
2     in getPorts(output portIDs: list);
3     in getWeavings(output weavingIDs: list);

4     in getAspectProperties(aspectID: string,
            output aspect: object, output linkedPorts: list,
            output definedWeavings: list);
5     in getPortProperties(portID: string, output port: object,
            output linkedAspect: string);
6     in getWeavingProperties(weavingID:string,
            output sourceAspectID: string,
            output weavingType: string,
            output targetAspectID: string);

7     in getStatus(elementID: string, output status: string);
8     in getElementsOfStatus(status: string,
            output elemIDList: list);
   ...
End_InstanceMonitoring_Aspect;
```

Figure 7.25. Services provided by the InstanceMonitoring aspect

Finally, services 7 and 8 return the status of a structural part. This allows us to know if a part is ready to be stopped or by the contrary is busy (e.g. processing a service request). The different execution statuses are the following (for more details about the safe stopping of running instances, see section 3.4):

- *Idle*. The structural part is waiting for new requests. In case of an aspect, no service is being processed or pending to be executed. In case of a weaving or a port, this means that no service is currently in the middle of an interception or being forwarded to an aspect, respectively.
- *Active*. A service request is being processed (in case of an aspect), intercepted (in case of a weaving), or forwarded (in case of a port). That is, the structural part is in the middle of a service transaction.
- *Blocked*. The structural part is in a consistent state and ready to apply runtime changes. Pending services, interceptions, or forwardings have been finished safely. New requests are accepted, but they are queued until the structural part status is changed to *Active*.
- *Blocking*. This is a transitional status from the *Active* status (executing/intercepting/forwarding a service) to *Blocked* status (not executing services and ready for reconfiguration). The element waits until the Tranquillity (Vandewoude et al., 2007) or Quiescence (Kramer & Magee, 1990) criterion is achieved. These criteria

guarantee that the structural part has reached a consistent state (tranquillity causes less disruption than quiescence).

- *Unknown*. The structural part is not responding because an error has occurred. This information can be used to provide fault-tolerance mechanisms.

In short, the *InstanceMonitoring* aspect has access to the low-level internal details of an instance at runtime (memory pointers, execution structures, runtime status). It filters and abstracts this low-level information to provide only the necessary data required by evolution processes.

Instance Effector aspect

The *InstanceEffector* aspect implements the low-level mechanisms that enable the modification of the internal elements of an instance at runtime. This aspect provides a set of atomic evolution services that hide the low-level mechanisms, so that changes can be described in terms of PRISMA elements: aspects, ports and weavings. The evolution services are simple because they do not take into account the status (i.e. whether the part to change is quiesced or not) and/or the relations with other instance parts. These evolution services must be correctly coordinated to carry out a safe transformation of the instance structure.

For instance, Figure 7.26 shows the most relevant services provided by the *InstanceEffector* aspect:

```
InstanceEffector Aspect
   ...
   Services
      // *** Services for Safe Stopping ***
1     in StopPart(elemID: string);    // Reach a Quiescent status.
2     in StartPart(elemID: string);    // Reach an Active status.

      // *** Atomic change operations ***
3     in CreateAspect(aspectTypeName: string, initParams: string,
            output aspectID: string);
4     in DestroyAspect(aspectID: string);
5     in CreatePort(name: string, interface: string,
            playedRole: string, output portID: string);
6     in DestroyPort(portID: string);
7     in CreateWeaving(sourceAspectID:string, sourceMethod:string,
            sourceParameters: list, weavingType: string,
            targetAspectID: string, targetMethod: string,
            targetParameters: list, transfFunctions: list,
            output weavingID: string);
8     in DestroyWeaving(weavingID: string);

      // Services for aspect updatings
9     in SerializeAspectState(aspectID: string,
            output state: string);
10     in UpdateStateStructure(oldAspectType: string,
            oldState: string, newAspectType: string,
            newRequiredValues: list,
            output transformedState: string);
11     in UnserializeAspectState(aspectType: string,
            serializedState: string, output aspect: object);
   ...
End_InstanceEffector_Aspect;
```

Figure 7.26. Services provided by the InstanceEffector aspect

Services 1 and 2 (*StopPart*, *StarPart*) manage the safe stopping and restarting of each one of the structural parts. Since ports and weavings are parts that enable the interaction among other parts (i.e. aspects), their stopping means that they do not forward/intercept service requests; these service requests are queued for their later processing. In the case of aspects, since they are stateful parts, their stopping means that running service requests are finished safely, and that new incoming requests (which involve new transactions) are postponed. In the implementation of these stopping services, the quiescence criterion is taken into account.

Services 3 to 8 (i.e. *createAspect*, *destroyAspect*, *createPort*, etc.) perform changes on the instance structure in memory: they allocate new memory space for new aspects, they add new services to the method description table when new ports are added, they introduce interceptions among existing methods when a new weaving is added, etc. However, the execution of these services does not take into account neither the dependencies among the structural parts when applying changes, nor if they are ready to be evolved. These services directly change the instance structure; if the instance has not been safely stopped before, then inconsistencies may be introduced in the instance state due to the execution of these services.

Note that a replacement operation is not explicitly provided. The replacement operation of stateful parts (i.e. aspects) is provided through the services 9 to 11. The service *SerializeAspectState* returns the internal state of an aspect in a type-dependent structure encoded as a string. The service *UpdateStateStructure* converts a previously serialized state of an aspect to the data structures of a new aspect type. Finally the service *UnserializeAspectState* creates an aspect from a previously serialized state. Thus, the replacement operation is performed through the consecutive execution of these services, which results in the migration of the old aspect state to a new aspect. These services are implemented in each aspect type and should be automatically generated for each aspect. Additional details about how to address the migration of state have been covered in section 6.4.4.3 (page 217), so they are not described again.

The implementation of the evolution services provided by this aspect is technology-dependent: depending on the technology selected and how the instance execution model has been implemented, the dynamic updating mechanisms to use will be different. However, the importance here is not the implementation of these mechanisms, but the identification of the minimum services required to support the reconfiguration process without the need to include low-level details.

In the case of our PRISMA implementation, the main challenges faced up are the management of running processes that are concurrently executing (i.e. one per aspect), and the decomposition of the instance structure into its parts (i.e. aspects, weavings and ports). On the one hand, the management of concurrent running processes has been managed by the development of an execution model that allows the asynchronous execution of services. Thus, when a stop is requested, all the incoming service requests are queued and postponed. The instance will be ready to evolve when the services that are being processed finish their computations. On the other hand, the decomposition of the instance structure has been performed by means of dynamic linking strategies. The reference to each structural part is available to the evolution mechanisms. When a structural part has been stopped, it can be safely removed or replaced from memory by unlinking it from other structural parts and by linking the new part to the other structural parts. For specific details, the reader is referred to (Costa-Soria, 2005), (Pérez et al., 2005a) and (Millán-Belda, 2006).

In short, the *InstanceEffector* aspect essentially allows an instance to: (i) isolate its structural parts (i.e. the entities and relations that compose the type); (ii) stop the running transactions involving these parts; (iii) change or replace these parts; and (iii) reassembly again these parts inside the instance boundaries. This is performed by abstracting the low-level details to other high-level coordination structures (i.e. the *InstanceEvolutionPlanning* aspect and the type meta-instance).

7.3.6 Summary of the evolution process

Finally, to provide a summary of the entire dynamic evolution process, this section describes the execution of a type evolution scenario, by means of the reflective infrastructure described in this chapter.

The scenario presented in section 5.5.2 (see page 154) describes that, after the delivery of the Agrobot to its final users (i.e. the farmers), a critical update is required: the updating of the image processing algorithms. These algorithms are located in the *ImageProcCard* and *ImageProcSoftware* component types. Since the former component is a hardware element that cannot be remotely changed, the maintenance team has decided to update only the latter, the *ImageProcSoftware*, which is automatically instantiated when needed. This update must be propagated and installed transparently, without disrupting the operations that the robot is carrying on (i.e. at runtime), and without any direct user operation (i.e. remotely, in an automated way). This dynamic, semi-automated updating is supported by means of the evolution infrastructure described in this chapter, which is described below.

The type evolution infrastructure defines three steps that must be followed to dynamically evolve an architectural type:

(i) Get a reification of the type to evolve,
(ii) Manipulate the reification provided by adding, removing or replacing elements of the type specification, and
(iii) Reflect the edited reification to the type meta-level.

Thus, to dynamically update the image processing algorithms of the Agrobot, the maintenance team should send to each Agrobot unit the following set of updating instructions (see Figure 7.27):

1) A set of instructions to get the reification of the type to evolve. The type to be evolved is the *ImageProcSoftware* type, which is imported by the *VisionSystem* type of the Agrobot. To get the reification of the *ImageProcSoftware* type, first its type reference (i.e. a pointer to the type meta-instance) must be obtained. This can only be done by going down through the hierarchy of compositions: from the top-level composition (the Agrobot architecture), to the level where the *ImagProcSoftware* type is imported (the VisionSystem architecture). This is shown in Figure 7.27. First, the reification of the Agrobot is obtained. Next, from the list of its architectural types, the reference to the type *VisionSystem* is obtained and reified. Third, from the list of the architectural types that the *VisionSystem* is composed of, the reference to the type *ImageProcSoftware* is obtained. Finally, the fourth operation shows *ImageProcSoftware* type is reified.

2) Edition of the reification with the set of desired changes. In this case, the change to perform is to replace the aspect *ImageProcSwController*, which encodes the image processing algorithms, by a new version: the aspect *ImageProcSwControllerV2*. This is shown in Figure 7.27, 5th operation.

3) Reflection of the changes back to the type. Once the changes to the reification have been completed, the set of changes are sent back to its type to initiate a dynamic evolution process. This is performed by the last operation of Figure 7.27. The reflection process also provides the set of evolution parameters (e.g. the strategy chosen to evolve instances, the timeout to evolve, etc.), which in this case are set to the default values (see page 295 for more details).

```
// Obtaining the reference of the type to evolve
Agrobot.Reify(out agrobotReification);
agrobotReification.architecturalTypeList["VisionSystem"].
     typeDefinition.Reify(visionSystemReification);
imageProcSoftwareTypeRef = visionSystemReification.
     architecturalTypeList["ImageProcSoftware"].typeDefinition

// Reification of the type to evolve
imageProcSoftwareTypeRef.Reify(out imageProcSWReification);

// Modification of the type to evolve: one aspect is replaced
imageProcSWReification.ReplaceAspect("ImageProcSwController",
     ImageProcSwControllerV2, "");

// Reflection of the changes, with the default evolution values
imageProcSoftwareTypeRef.Reflect(imageProcSWReification,
     new EvolutionPolicy());
```

Figure 7.27. Sequence of evolution instructions performed to dynamically update the type ImageProcSoftware

As a result of the *Reflect* operation, the type is dynamically changed: a new type version is automatically generated, and its existing instances are progressively transformed to the new version. This is performed by the type evolution infrastructure without any further intervention of the evolution agent, in a totally automatized and transparent way.

From an internal point of view, when an edited reification is reflected to the type meta-level, different actions are performed at both the type-level and the instance-level. At the type-level, first the instance-creation and type-reification mechanisms are blocked, to avoid the creation of instances or reification of the type version that is going to be evolved. Next, if evolution preconditions are satisfied, then a new type version is generated: this involves the generation and compilation of the executable code of the new type version and the replacement of the old one. If this process finishes successfully, the instance-creation and type-reification mechanisms are unblocked, so new instantiations and type reifications would be performed according to the new type version.

Then, the changes introduced by the new type version are asynchronously propagated to the instance-level. At the instance-level, each instance receives the set of changes and stores it for later management. According to different, local conditions, when the instance decides to start an evolution process, then: (i) it processes the set of evolution requests received, evaluating its version conformance, (ii) reaches a quiescent status to finish consistently running transactions, (iii) modify its structure dynamically, according to the set of evolution steps that have been received from the type-level, and (iv) if it is possible, it migrates the old state to the new data structures introduced by the new type version.

7.4 Conclusions & further works

This chapter has presented another perspective of this work: the support for *Dynamic Evolution of Architectural Types*. This work provides four contributions to the state-of-the-art on type evolutions: (i) type-oriented reflective evolutions, (ii) support for asynchronous evolutions, (iii) evolution of instances through transformations, and (iv) separation of evolution concerns through aspects. This section summarizes these contributions, the remaining works, and the related publications.

7.4.1 Conclusions

This chapter has described a novel reflection-based approach for supporting the dynamic, asynchronous evolution of architectural types. This approach combines the following features in an innovative way:

Type-oriented evolutions. Evolution support is considered as a feature of each architectural type, so evolution mechanisms are only integrated in these types that actually require them. This avoids that non-evolvable types would be affected by the execution overhead introduced by evolution mechanisms, and removes the need of a centralized evolution manager, which could be a single-point-of-failure (if it fails, evolution will not be supported).

Reflective descriptions. Evolution is provided through reflection: each evolvable architectural type is able to obtain (and provide) an editable description of itself, and to reflect the changes on its description. Reflective capabilities are provided by means of type meta-instances, which contain the meta-level representation of a type and are able to regenerate the executable code when this meta-representation is changed. Thus, a type meta-instance makes a type self-described and self-evolvable. This also simplifies the description of evolution.

Asynchronous evolution model. Evolution is performed asynchronously: each architectural type and its instances evolve at different rates. This has been supported by distributing the evolution mechanisms among the type-level and the instance-level and by implementing version-management strategies. This benefits the propagation of changes in distributed and/or mobile systems: a type can be evolved independently of its instances, which may receive and integrate the changes later, when they are ready. This gives a great level of flexibility to the evolution process: an instance may postpone its evolution until it reaches a quiescent status, or until its context is prepared for evolution (i.e. it is semantically compatible with the new version).

Instance evolution through transformations. The evolution of instances is performed by means of the decomposition of the internal structure of an instance and its incremental transformation, by adding/removing the structural elements that have been added/removed from the type specification. This benefits the transparent evolution of instances, without the need of recreating again the external links or migrating the state of the instance. In addition, for these cases in which a type could not be decomposed, support for instance migration has been also provided (e.g. for the replacement of aspects).

Separation of concerns through Aspect-Oriented techniques. The evolution mechanisms have been separated into different concerns, which in turn are distributed among type-level and instance-level concerns. These concerns have been encapsulated as aspects, with a special emphasis of making these aspects independent of each other (through weavings), to increase reuse and maintenance. Type-level aspects address the evolution of both the specification and executable code, whereas instance-level aspects address the evolution of the internal structure of each instance. These aspects are reused by all the evolvable architectural types and its instances, except the Builder aspect, which contains the specific executable code of each type.

It is important to emphasize that the approach presented is very flexible: evolution processes can be time-constrained and/or applied to a subset of instances. This allows us to either force or abort the integration of changes if they are not performed in a certain time. In addition, instances can be excluded from evolutions if it is needed (e.g. due to semantic incompatibility), without losing the possibility of reintegrating them in the evolution path later.

7.4.2 Further works

A feature that has been left to future work is the definition of constraints for the evolution of architectural types. Currently, our proposal is generic enough to allow the dynamic evolution of any element of a type specification. However, in certain cases it may be interesting to limit which parts of the type can be evolved or not. An interesting starting point from where to start could be the work of (Rutle, 2010), which addresses the integration of constraints in meta-modelling (i.e. in the type definition), and the constraint-aware model transformation.

Concerning the PRISMA implementation, an earlier prototype of the type evolution infrastructure has been developed in .NET (Hervás-Muñoz, 2009). However, it remains the integration of this evolution infrastructure in the PRISMA Case Tool, to automatically include the evolution mechanisms in the generated code of evolvable types. Currently, the evolution mechanisms are added after the generation process, by manually editing the generated code to include the libraries containing the type-level and instance-level aspects. However, the principles and knowledge are well established from previous works (Guillén-Martín, 2007), (Pérez et al., 2007), (Pérez et al., 2007a), so its implementation is straightforward.

Further works remain, such as addressing the impact of evolutions on the interactions with the adjacent architectural types. Some authors have addressed this issue by means of the dynamic generation of adaptors that act as mediators among the existing instances and the replaced (or evolved) ones (Cámara et al., 2008). We have also started addressing this issue from a different perspective, by means of the coordination among the evolution mechanisms and other elements, in a non-intrusive way. Since this is still in an early stage and some issues remain opened, this has been intentionally left outside this thesis. The early findings can be found in (Costa-Soria & Heckel, 2010), (Costa-Soria et al., 2011).

7.4.3 Results

The work related to the definition of the dynamic evolution of architectural types has been presented in the following publications:

- **Costa-Soria C.**, Hervás-Muñoz D., Pérez J., Carsí J.A. A *Reflective Approach for Supporting the Dynamic Evolution of Component Types*. In proc. of: 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'09), pp. 301-310. Potsdam, Germany, 2-4 June 2009.
- **Costa C.**, Pérez J., Carsí J.A. *Managing Dynamic Evolution of Architectural Types*. Morrison, R., Balasubramaniam, D., Falkner, K. (eds.): 2nd European Conference on Software Architecture (ECSA'08). Lecture Notes on Computer Science, vol. 5292, pp. 281-289. Springer, Heidelberg, October 2008.
- **Costa C.**, Pérez J., Carsí J.A. *Soporte a la Evolución Dinámica de Tipos Arquitectónicos*. In proc. of: 1st Workshop on Autonomic and Self-Adaptive

Systems (WASELF'08). Gijón, Spain, October 2008 *(in Spanish)*.

- **Costa C.**, Pérez J., Carsí J.A. *Dynamic Adaptation of Aspect-Oriented Components*. Schmidt, H.W., Crnkovic, I., Heineman, G.T., Stafford, J.A. (eds.): 10th International Symposium on Component-Based Software Engineering (CBSE'07). Lecture Notes on Computer Science, vol. 4608, pp. 49-65. Springer, Heidelberg, July 2007.

CHAPTER VIII

DESCRIPTION OF THE EVOLUTION SEMANTICS

8.1 Introduction

This chapter describes the semantics of asynchronous type evolution from a high-level of abstraction, by means of typed graph transformations. This formalism has been chosen because it naturally models both the system architecture and the asynchronous nature of its evolution. This has been realized by describing the observed behaviour of evolution services (both at the type-level and at the instance-level), and by using an architecture-based concrete syntax, which is more concise and easier to understand than the graph-based abstract syntax. This has facilitated the identification and resolution of some issues related to asynchronous evolution, such as the management of type conformance, the order of evolution processes or the coherence of interactions.

This chapter is organized as follows. Section 8.2 presents the challenges posed by asynchronous evolution. Next, section 8.3 presents how evolution semantics has been described by means of typed graph transformations. Finally, section 8.4 presents the conclusions and further works.

8.2 Challenges of asynchronous evolution

As described in section 7.3.4 (see page 266), asynchronous evolution takes a step forward from synchronous evolution. Asynchronous evolution of types allows us to introduce multiple (but ordered) change requests, deferring them until they can be effectively applied. For instance, this will allow different stakeholders to update different parts (i.e. types) of a system at different times, without taking into account the update of the running (and perhaps distributed) instances. However, these advantages make asynchronous evolution the most challenging.

Most of related works have been focused on the challenges posed by synchronous evolution:

- The adaptation of runtime data structures and code: (Segal & Frieder, 1993), (McKinley et al., 2004), or (Nicoara et al., 2008).
- The state consistency before and after a dynamic change: (Kramer & Magee, 1990), (Gomaa & Hussein, 2004), (Vandewoude et al., 2007)
- The migration of the state: (Ritzau & Andersson, 2000), (Vandewoude & Berbers, 2005)

Asynchronous evolution must also cope with these challenges, but in addition it poses new challenges:

1. *Type conformance*. Since types evolve at different rates with respect to their instances, it is then difficult to check if the instance-level is conformant to the type-level.
2. *Version management*. Different evolutions of a single type entail the existence and management of different versions of this type at runtime (at least where there are running instances of such type), which adds more complexity to the process.
3. *Order of evolution processes*. In such a case where a type could require a high rate of evolutions, it could happen that an instance might have two or more pending evolutions. In this case, the order of pending evolutions must be preserved. This is important in distributed systems, where instances may receive newest evolution changes before the older ones.
4. *Coherence of interactions*. Interactions among instances that are in different versions can produce incorrect behaviours. The context where instances evolve is also important.

These challenges have been taken into account when describing the semantics of the asynchronous evolution, which is presented in the next section.

8.3 Evolution semantics

An evolution process comprises the execution of several evolution operations, which are of different kinds (i.e. additions, removals or updates) and focused on changing a specific part of an architectural type (e.g. in case of composite types: its architectural types, its ports, its attachments, its bindings, ...).

8.3.1 Specification of evolution processes

As described in Chapter 7, an (asynchronous) dynamic evolution process is triggered when a meta-service called *Reflection*, provided by each evolvable architectural type, is invoked (see page 261). This service requires a new specification of the type to evolve (i.e. a reification), which is described in terms of modifications on the current type specification: *additions*, *removals* or *updates*.

The set of modifications that can be performed on an architectural type is defined by its metamodel[44]. According to the PRISMA metamodel (see section 2.4, page 47) and the characteristics of the System architectural element (i.e. the parts that a composite component consists of), 12 evolution operations have been defined (see Figure 8.1). Each operation involves runtime changes both at type-level and instance-level.

```
AddArchitecturalType(AEName, type, minCard, maxCard);
AddAttachmentType(attName, sourceAE, srcPort, srcMinCard,
        srcMaxCard, targetAE, trgPort, trgMinCard, trgMaxCard);
AddBindingType(bindName, sysPort, targetAE, trgMinCard, trgMaxCard);
AddPort(portName, interface);
RemoveArchitecturalType(AEName);
RemoveAttachmentType(attName);
RemoveBindingType(bindName);
RemovePort(sysPortName);
UpdateArchitecturalType(oldType, newType, minCard, maxCard);
UpdateArchitecturalTypeCard(AEName, newMinCard, newMaxCard);
UpdateAttachmentType(attName, srcMinCard, srcMaxCard, trgMinCard,
        trgMaxCard);
UpdateBindingType(bindName, trgMinCard, trgMaxCard);
```

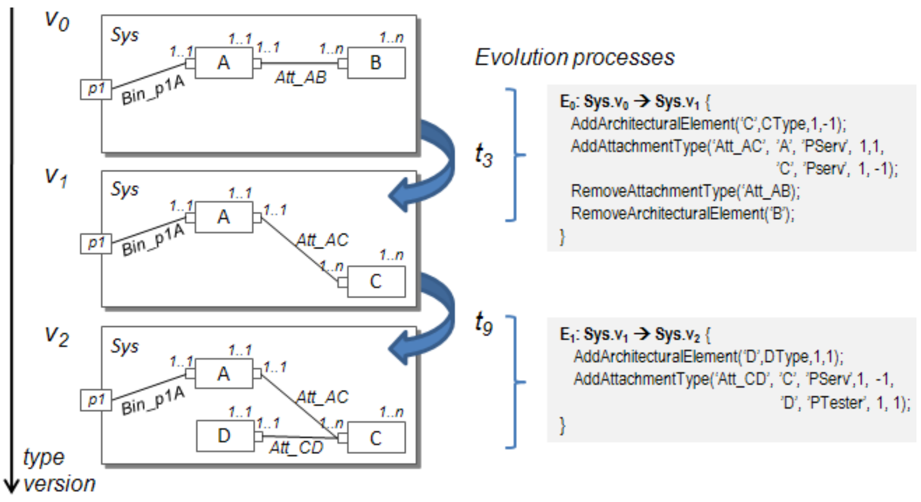
Figure 8.1. Evolution operations over Composite types

For instance, Figure 8.2 shows how a System type, named *Sys*, is evolved by two evolution processes from the initial version, v_0, to versions v_1 and v_2. The first evolution process (see E_0: $Sys.v_0 \rightarrow Sys.v_1$ in Figure 8.2), started in time *t3*, performs the following operations: (i) introduces a new architectural type C and constrains its minimum and maximum cardinality to *1* and *n*, respectively; (ii) attaches this type (through the attachment type called 'A-C') to the port '*pA*' of type *A*, with the cardinalities C→A[1..1], A→C[1..n]; (iii)

[44]A metamodel describes how an architectural type is defined: the parts it is composed of, their relationships and attributes.

removes the type *B*; and (iv) removes the attachment from *A* to *B*. The resulting Sys type version, v_1, is shown in the left.

The second evolution process (see E_1: Sys.v_1 → Sys.v_2 in Figure 8.2), started in *t9*, performs two operations: (i) introduces a new type *D*; and (ii) attaches it to to the type C (which was added in the previous evolution process). The resulting Sys type version, v_2, is shown in the left.



```
E0: Sys.v0 → Sys.v1 {
  AddArchitecturalElement('C',CType,1,-1);
  AddAttachmentType('Att_AC', 'A', 'PServ', 1,1,
                              'C', 'Pserv', 1, -1);
  RemoveAttachmentType('Att_AB);
  RemoveArchitecturalElement('B');
}

E1: Sys.v1 → Sys.v2 {
  AddArchitecturalElement('D',DType,1,1);
  AddAttachmentType('Att_CD', 'C', 'PServ',1, -1,
                              'D', 'PTester', 1, 1);
}
```

Figure 8.2. Example of two evolution processes

The successful execution of each evolution process creates a new version (i.e. a new specification) of the architectural type being evolved. Type versions are identified from each other by means of a version number, which is increased each time an evolution process is applied on the type successfully. Each instance also contains an attribute which keeps track of the type version that it is currently an instance of. This is used to control that only the evolution operations leading to the next type version (from the instance perspective) are taken into account, thus preserving the order of evolutions.

Next section presents how different type versions are managed, by means of evolution tags.

8.3.2 Version management: Evolution Tags

A contribution of the approach presented in this thesis is that *only one single type specification is needed to describe both the current and previous type versions*. This is possible because each change that is performed on an architectural type is also reflected on its specification, by means of evolution tags.

An **evolution tag**:

- Describes the kind of evolution operation that has been performed: an addition (+), a removal (–), or an update (U).
- Is linked to the element of a type specification on which the evolution operation has been performed (e.g. in the case of a composite component: architectural elements, ports, attachments, bindings, constraints, constructors and destructors)
- Indicates the type version where the evolution operation took place.

Thus, the specification of a type which makes use of these evolution tags can describe not only the structure (and behaviour) of a type, but also the history of its evolutions over time.

Figure 8.3 and Figure 8.4 show, respectively, a graphical and textual specification of a System type that includes the description of its evolutions over time by means of evolution tags. In particular, these figures describe the resulting System type *Sys* after the execution of the evolution processes presented in Figure 8.2. For instance, the tag *[+,1]* near the declaration of the type C tells us that the type C has been imported in version 1 of *Sys*.

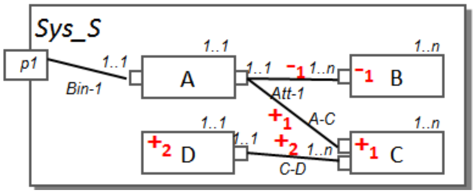


Figure 8.3. Graphical specification of a type tagged with Evolution tags

```
System Sys
   Ports
      P1 : interface1;
   End_Ports;

   Import Architectural Elements
      A(1,1),
      B(1,n)[-,1],
      C(1,n)[+,1],
      D(1,1)[+,2];

   Attachments
      Att_AB: A.PServ(1,1) <--> B.PServ(1,n);   [-,1]
      Att_AC: A.PServ(1,1) <--> C.PServ(1,n);   [+,1]
      Att_CD: C.PServ(1,n) <--> D.PTester(1,1);[+,2]
   End_Attachments;

   Bindings
      Bin_p1A: P1(1,1) <--> A.PClient(1,1);
   End_Bindings;
End_System Sys;
```

Figure 8.4. Textual specification of a type tagged with Evolution tags

Note that, over time, such tagged specification may become difficult to read or interpret, as the history of evolutions grows. However, it must be taken into account that evolution tags are targeted for internal use of type descriptions: they allow us to embed both the definition of an architectural type and its evolution in a single place, which eases its management.

To get the details of a specific type version (e.g. to be returned to an evolution agent), it has been defined the function *GetTypeVersionSpecification(T, v)*. This function "filters" all the information that is contained in a tagged specification and returns only the elements that belong to a specific type version (see Figure 8.5).

$$
\begin{aligned}
&GetTypeVersionSpecification(T, v) = \\
&\forall e, e \in elements(T), \forall tag, tag \in e.EvolutionTags: \\
&\qquad (tag = \emptyset) \lor \\
&\qquad (tag.Type = Added \land tag.Version \leq v) \lor \\
&\qquad (tag.Type = Removed \land tag.Version > v) \lor \\
&\qquad (tag.Type = Updated \land tag.Version > v)
\end{aligned}
$$

Figure 8.5. Function GetTypeVersionSpecification

From the set of all elements that the tagged specification of a type *T* consists of (i.e. *elements(T)*), this function returns only these ones that were defined at a version '*v*'. That is:

- *Unchanged elements*. That is, elements of the type T that do not have any evolution tag.
- *Elements defined both in version 'v' and in previous versions*. That is:
  - *Elements that have been added in versions<=v*. Those elements that have been added subsequently (i.e. elements which *tag.Version>v*) are excluded from the resulting set: they do not exist yet in the type version that it is required.
  - *Elements that have **not** been removed or updated in versions<=v*. If an element is tagged with a "*Removed*" or "*Updated*" evolution tag, it will be selected if and only if this evolution tag belongs to future versions. The element that replaces another element is considered as an addition, so it is included in the previous case.

The function *GetTypeVersionSpecification* has been used in the description of the evolution semantics to constrain the set of pending evolution operations that are visible to each composite instance. Each composite instance is only provided with the set of changes that it should apply to be transformed to the subsequent type version. This makes each instance to progressively evolve from one type version to the following, thus preserving the order of evolutions.

Next, the semantics of the evolution operations is described, which will illustrate how evolution tags play a central role in communicating each composite instance with the set of changes to be applied.

8.3.3 Formalisation of the evolution operations

The semantics of the evolution operations corresponding to composite architectural types has been formalised by means of typed graph transformations (Heckel, 2006). Graph transformations combine the idea of graphs as a modelling paradigm with a rule-based approach to specify the evolution of graphs. They are supported by an established mathematical theory and a variety of tools for its execution and analysis (Ehrig et al., 2006). The main reasons to choose typed graph transformations as the basis for the formalisation are the following:

(i) Software architectures can be easily formalised as graphs, as shown in other works (Hirsch et al., 1998), (Wermelinger et al., 2001);

(ii) Graph transformations are asynchronous, i.e., each rule can be applied once its preconditions are satisfied, which benefits the formalisation of asynchronous evolution; and

(iii) Typed graphs capture the relation among types and instances, required to model the evolution both at the type-level and instance-level.

In this work, typed graph transformations have been used to describe the *observed behaviour of evolution operations*: i.e. how a System type (i.e. a PRISMA composite type) and its instances change as a result of the execution of evolution operations.

The description of the evolution semantics has produced near 40 typed graph transformation rules, concerned with the management of evolutions at the type-level, the instance-level, or both. However, only the most representative transformation rules are described in this chapter. They are enough to illustrate how the different challenges related to asynchronous evolution are addressed and how evolution is described. This is possible because each transformation rule is self-contained and can be understood without requiring the description of the complete set of rules.

Typed graph transformation rules are presented using an *architecture-based concrete syntax*, which is more concise and easier to understand than the *graph-based abstract syntax*. Instead of using only vertices and nodes (as provided by the graph-based abstract syntax), a concrete syntax allows describing the same behaviour using concepts closer to the area of software architectures (i.e. components, ports, attachments, etc.), but extended with concepts required to deal with asynchronous evolution.

Next, these architecture-based transformation rules are presented, and afterwards, their mapping to the graph-based abstract syntax.

8.3.3.1 Architecture-based Concrete Syntax

Among the different evolution operations that have been defined for evolving PRISMA composites, only two of them have been selected: *AddArchitecturalType* and *UpdateArchitecturalType*. The former has been selected due to the simplicity of its transformation rules, which allows the reader to get introduced in the semantics of graph transformations. The latter has been selected because it copes with one of the most important issues of dynamic evolution: the coherence of interactions and the migration of state after replacements. Next the modelling of these evolution operations are described in detail.

Addition of a new architectural type

The addition of a new architectural type to a composite type (i.e. a System) is performed by executing the *AddArchitecturalType* evolution operation. This operation modifies the composition of a System type to add a new, unforeseen, architectural type at runtime. This operation requires four parameters:

- *AEName*, the alias of the type to add;
- *type*, the executable code of the type to add;
- *minCard*, the minimum number of instances of this type that must exist in each system instance; and
- *maxCard*, the maximum number of instances that can exist in each system instance (i.e. in each Configuration).

The semantics of this evolution operation is modelled by means of different transformation rules which operate at both the type-level and the instance-level.

♦ Type-level Rules: AddArchitecturalType

The rule *AddArchitecturalType*, which operates at the type-level, describes how a composite type (i.e. a System) is changed to introduce a new architectural type in its composition. This rule is provided with the same parameters that are provided to the evolution operation *AddArchitecturalType*, described above. A graph transformation rule consists of a left hand side, which describes the context where the rule can be executed, and a right hand side, which describes the result of executing the rule.

The left hand side of the rule R3 (see Figure 8.6, left side) defines the context: the System type to modify, and the set of its instances. The System type to modify is represented as the top-left component named 'S', and matches with the type where the evolution rule has been invoked. The set of its instances are represented as a multiobject (i.e. a collection of elements) named 'S1' at the bottom-left. There is a condition that must be satisfied to execute this rule: the architectural element to add (i.e. *AEName*) must not already be present in the pattern defined by the System type. This is represented graphically as a component called 'AEName' inside 'S' which is crossed out.

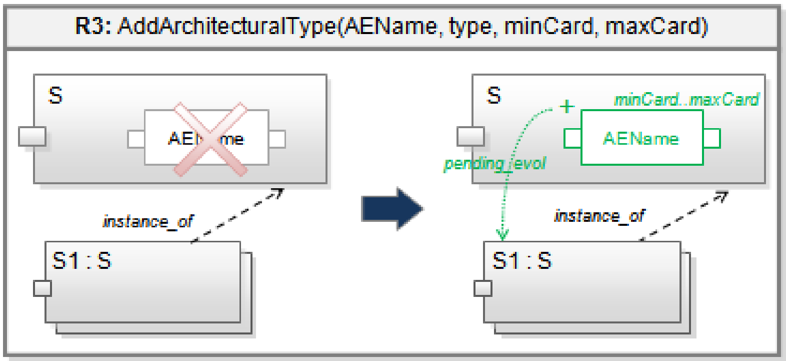


Figure 8.6. Rule R3: AddArchitecturalType

Note that all the elements, 'S', 'S1', and 'AEName' are variables whose values are set when the rule is applied on a specific element. For instance, if the following is executed:

```
VisionSystem.AddArchitecturalType(ImageProcCard,…)
```

Then, 'S' would be set to *VisionSystem*, 'S1' would be set to {*RightCamera*, *LeftCamera*} and 'AEName' to *ImageProcCard*.

The right hand side of the rule R3 (see Figure 8.6, right side) describes the result: the pattern of the System type represented by 'S' is modified to include a new architectural type, identified by *AEName*. This new type can be instantiated in each one of the System instances, but only while satisfying the *minCard* and *maxCard* constraints.

As described in section 8.3.1, the specification of each architectural type keeps track of the changes performed on it, to describe its evolution over time and to control the asynchronous evolution of its instances. This process is also described in transformation rules. When the rule R3 is executed, a new *add* evolution tag (represented by the symbol '+') is created and attached to the new architectural element (see Figure 8.6, right side). This evolution tag is initialized with the current version number of the System type (whenever a new evolution process is started, the version number of the type is increased; the new evolved type is identified by this version number). This tag is also provided with a link to every instance of the type that must be evolved to the new type specification (remember that instances can be optionally excluded from evolutions). This relationship is reflected in R3 with the '*pending_evol*' link to the set of instances. The use of this link is described later.

Overall, the result of this transformation rule is that the specification of a System type is changed, by introducing a new architectural type and an addition tag that describes in which version this event took place. Next, type-level changes are propagated to the instance-level. This is performed through instance-level rules.

♦ Instance-Level Rules

Instance-level rules carry out asynchronously the changes that have been introduced at the type-level. For instance, after the execution of the type-level rule *AddArchitecturalType* on a composite type, three instance-level rules follow on each one of its composite instances:

(i) *CreateArchitecturalElement*, which creates instances of the type that has been added;
(ii) *ActivateInstantiatedArchElements*, which initiates the execution of the new instances when the conditions (i.e. the minimum cardinality) defined by the composite type are fulfilled; and
(iii) *AdvanceInstanceToNewVersion*, which promotes a composite instance to a new version when all the requirements of the new version have been fulfilled.

Instance-level rules can be deterministic or non-deterministic. Deterministic rules are those that are automatically triggered once some conditions hold, whereas non-deterministic rules are those that require some kind of user information to be executed. For instance, the rules *ActivateInstantiatedArchElements* and *AdvanceInstanceToNewVersion* are deterministic, because they are automatically triggered when the conditions they define are fulfilled. By the contrary, the rule *CreateArchitecturalElement* is non-deterministic, because it requires that the user provides the initialisation parameters of the new instance. Next these rules are described in detail.

Rule R20: CreateArchitecturalElement

The rule *CreateArchitecturalElement* describes the instantiation of an architectural type inside a composite instance. This rule requires three parameters: (i) *AEType*, the name of the architectural type which to create an instance; (ii) *id*, a unique identifier of the instance to create; and (iii) *params*, the parameters required to create an instance of the type *AEType*.

This rule can be executed independently, as part of a reconfiguration process (i.e. to create new instances according to the constraints defined in the pattern of the composite type), or as part of a type-level evolution process. In the first case, the parameters of the rule would be provided by the reconfiguration agent (either proactive specifications or an external user, see 3.3.4.2, page 72). In the second case, the parameters of the rule (mostly *id* and *params*) must be provided by the evolution agent, i.e. the initiator of the type-level evolution process. For this reason, this rule is non-deterministic.

The semantics of this rule is described in Figure 8.7. In this rule, "*S1*" is a Configuration (i.e. the name given in the PRISMA ADL to composite instances), and its type is the System "*S*" (i.e. the name given in the PRISMA ADL to composite types). The left-hand side of the rule describes which conditions must be satisfied to allow the instantiation of an architectural type, *AEType*, in the Configuration *S1* (the context where the reconfiguration service has been invoked). These conditions are the following:

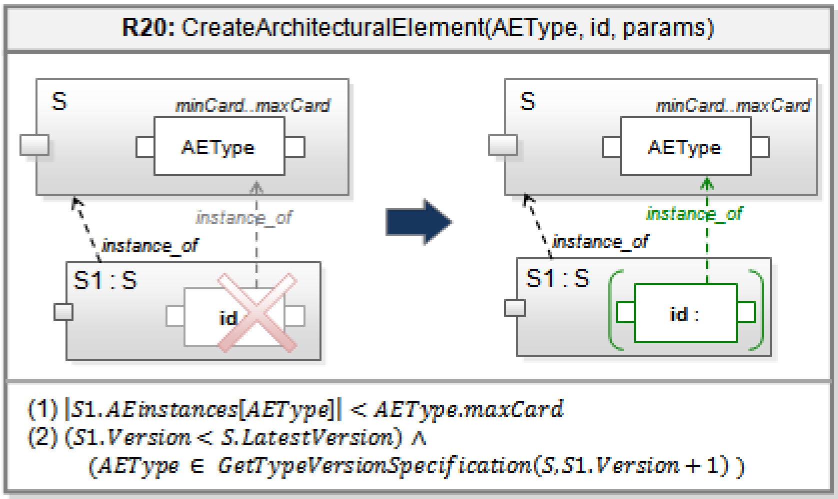
(i) The architectural type *AEType* must be defined in the type of *S1*: *AEType* exists in the pattern defined by the composite type *S* (see Figure 8.7, top, in the box called *S*)

(ii) The composite instance *S1* cannot have another instance of *AEType* with the same identifier *id* (see Figure 8.7, red cross)

(iii) The number of instances of *AEType* created in the composite instance *S1* must be lower than the maximum cardinality *maxCard* defined in the composite type *S* (see Figure 8.7, condition 1).

(iv) If there are pending type-level evolutions (i.e. the version of the composite instance S1 is lower than the latest version of the composite type S, see Figure 8.7, condition 2), then the type *AEType* must be also included in the next version of the composite type S (see the second part of the condition 2 in the figure).

The last condition checks that the architectural element that is going to be instantiated has not been removed (or replaced by another one) from the composite type *S* (i.e. it has not been tagged for deletion or update). Or, if it has been removed, that it has not been done in the next version that the composite instance is pending to evolve (i.e. *S1.Version+1*). In addition, this condition also forbids the instantiation of architectural elements that have been added in future versions of the composite type S. Those elements that belong to type versions that are far from the subsequent version which a composite instance is evolving to (i.e. belonging to versions greater than *S1.Version+1*) are hidden to composite instances. Composite instances must evolve in an ordered way: first all the changes of the *S1.version+1* must be included, then the changes belonging to *S1.version+2*, etc. Thus, the rule takes into account not only the conformance of a Configuration to its System type, but also the convergence to the subsequent evolution of its System type.


R20: CreateArchitecturalElement(AEType, id, params)

(1) $|S1.AEinstances[AEType]| < AEType.maxCard$

(2) $(S1.Version < S.LatestVersion) \wedge$

$(AEType \in GetTypeVersionSpecification(S, S1.Version + 1))$

Figure 8.7. Rule R20 - CreateArchitecturalElement

The right-hand side of the rule describes how the *AEType* architectural element is instantiated in a Configuration (i.e. a composite instance). Each new instance created is left in a *quiescent* state by default, i.e. it cannot start or process any transaction (see page 74 for a complete definition of quiescence). The quiescent state is very important, and for this reason it has been represented in the rules by using the symbols '[' and ']' around the software artefact being quiescent (see Figure 8.7, right-hand side). The creation of instances stopped by default is to guarantee that the minimum cardinality constraint defined in the System type is satisfied first, before allowing the execution of these instances. Thus, several instances can be created, but they will not be started until their number would not be enough.

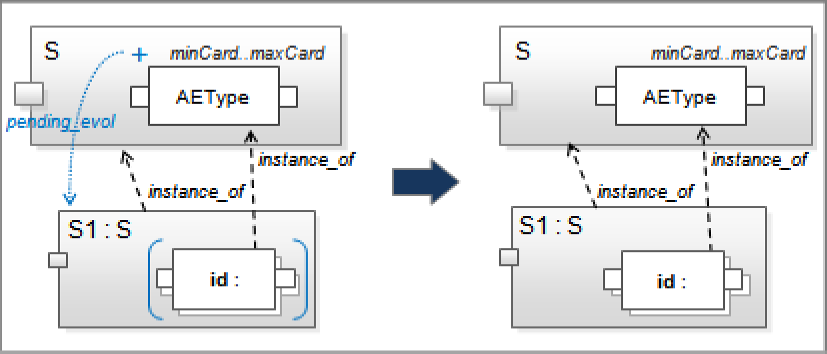
As a result of this transformation rule, the internal composition of a composite instance (i.e. its architecture) is changed to accommodate a new architectural instance, which is quiescent by default. This rule will be executed as many times as necessary until the number of the architectural instances created is equal or greater than the minimum cardinality. This is checked by the following rule, *ActivateInstantiatedArchElements*.

Rule R21: ActivateInstantiatedArchElements

The rule *ActivateInstantiatedArchElements* checks whether a composite instance has completed all the steps belonging to the evolution operation *AddArchitecturalElement* or not. That is, it checks that a composite instance: (i) has integrated the new architectural element in its composition, and (ii) has created the required number of instances of the new architectural element.

This rule (see Figure 8.8) is deterministic, and is automatically triggered for each composite instance (depicted as 'S1' in the figure) that fulfils the following conditions:

R21: ActivateInstantiatedArchElements



(1) $|S1.AEinstances[AEType]| \geq AEType.minCard$
(2) $(S1.Version < S.LatestVersion) \land$
$(AEType \in GetTypeVersionSpecification(S, S1.Version + 1))$

Figure 8.8. Rule R21 - ActivateInstantiatedArchElements

(i) Its composite type, *S*, imports an architectural element, *AEType*, that has been recently added (i.e. it is tagged with the symbol '+') AND that the composite instance has not integrated yet (i.e. it has a *pending_evol* link to the composite instance). This avoids that this rule could be activated to unblock instances that have been quiesced by another rules.

(ii) The composite instance, *S1*, has created one or more instances of the architectural element *AEType*. The set of instances is modelled by means of a multiobject identified by *id*, placed inside *S1*, and linked to the *AEType* element by means of a link *instance_of*.

(iii) All the instances of *AEType* are in a quiescent status. This is modelled by enclosing the multiobject *id* by the symbols '[' and ']'.

(iv) The composite instance *S1* has created as many instances of *AEType* as specified in the property *mincard* defined in its composite type, *S*. This is described by the condition 1 (see Figure 8.8, the first condition below the rule).

Finally, condition 2 is auxiliary: it guarantees that the specification of the composite type that is chosen by the rule conforms to the version that the composite instance is evolving to (i.e. *S1.version+1*).

As a result, the execution of this rule has two effects. One the one hand it unblocks (i.e. it starts the execution of) the set of instances of *AEType*. This is represented by removing from the right hand side the symbols '[' and ']' that were enclosing the multiobject *id* on the left hand side.

On the other hand, another effect of the rule is that it removes the link '*pending_evol*' among the addition tag (depicted as '+') and the composite instance *S1* where the rule has been applied. This means that the composite instance *S1* has integrated the new architectural element *AEType* in its structure, as specified by the composite type *S*. In other words, the composite instance has completed the evolution operation *AddArchitecturalElement*.

However, since an evolution process consists of several evolution operations, the completion of an evolution operation (as represented by this rule) does not mean that the composite instance has changed of type version. It has only completed *one* of the evolution operations that are pending. Only when all the pending evolution operations are realised, then the instance could be promoted to the next version (i.e. to increase the version of a composite instance). This is described by the following rule, *AdvanceInstanceToNextVersion*.

Rule R38: AdvanceInstanceToNextVersion

The rule *AdvanceInstanceToNextVersion* describes when a composite instance has finished an evolution process, i.e. it has integrated all the type-level changes leading to the subsequent type version. A composite instance has finished an evolution process when it has successfully applied all the evolution tags leading to the next type version. That is, given a composite instance *S1* that is conformant to a type version *v*, then it could be promoted to the next type version *v+1*, if and only if no evolution tag of version *v+1* exists with a '*pending_evol*' link to the composite instance *S1*. Recall that this link points to these composite instances that have still not applied an evolution tag.

This is modelled in the following way (see Figure 8.9, left-hand side): a multi-object named *Evol_Tag* models the set of evolution tags which fulfil two conditions: (i) they belong to type version *v+1* (i.e. the subsequent version of the composite instance which is being evaluated); and (ii) they have a '*pending_evol*' link to the composite instance being evaluated, *S1*. Since this multi-object is crossed out, this means that this rule can be only executed if the multi-object that fulfils the previous conditions is *empty*. This rule is automatically triggered for each composite instance which satisfies the previous condition.

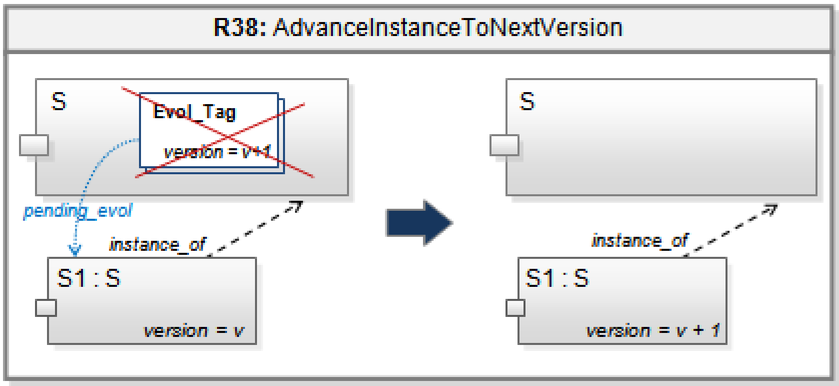


Figure 8.9. Rule R38 - AdvanceInstanceToNextVersion

The result of this rule (see Figure 8.9, right-hand side) is that the version of the composite instance on which this rule has been applied is increased: *S1.version=S1.version+1*.

From this moment, then the composite instance *S1* is **fully conformant** to version *v+1* of the composite type S. By contrast, note that during the asynchronous execution of an evolution process, instances are **partially conformant** to both their current type version and their subsequent type version. However, only when an instance is fully conformant to a type version, then it may start evolving to another version (if there are still pending evolution processes).

In addition, the 'pending_evol' link has an additional use: to monitor the evolution state of each instance at the type-level. By looking the oldest evolution tag (i.e. with the minor version) which is still linked to a composite instance, it lets us know: (i) the set of evolution operations a composite instance is currently involved in, and (ii) the version a composite instance currently is conformant to, by decreasing the version provided by the oldest evolution tag found.

Updating of Architectural Types

Finally, in order to fully understand our approach, the update operation is described here. This operation replaces a type version by another and updates its instances, keeping all their existing connections and their internal state. This is modelled by means of two transformation rules: *UpdateAEType* and *ReplaceAE*, which act at the type-level and the instance-level respectively. The context where these rules are executed are: the composite type (i.e. a System) that imports the type to update, and the composite instances (or Configurations) that have imported and instantiated the type to be updated.

♦ Rule R10: UpdateAEType

On the one hand, the *UpdateAEType* rule models the updating of a type version by a new one, keeping the existing interaction patterns. Let us recall that in the PRISMA model, the type-level defines the interaction patterns among types (i.e. what kind of interactions are allowed), and consequently, the interactions among their instances. When updating a type, the existing interaction patterns must be preserved to guarantee the coherence of interactions at the instance-level: the instances that were connected/attached to the instance to update must be able to interact with the same instance after the updating process.

The coherence of interactions can be only guaranteed by requiring the new type version to be syntactically and semantically compatible with existing interactions; otherwise, the update operation must not be performed. That is, compatibility evaluation when performing an update is focused on the interactions that the old type has, instead of focusing on the type itself. The reason is that a new type version, despite being incompatible with its previous version, could be semantically compatible with the types that were interacting with the previous version. This is the case when, in the context of an evolution process, the removal of *part of* the original functionality is required: a new version provides less functionality than the previous version (i.e. the new version is not compatible with the old one), and the interactions requiring this functionality have been removed from the initial set of interactions (i.e. the new version is compatible with the resulting set of interactions). If complete compatibility (or subtyping) of the new type with respect to the old type were required, we would never be allowed to remove unused functionality[45].

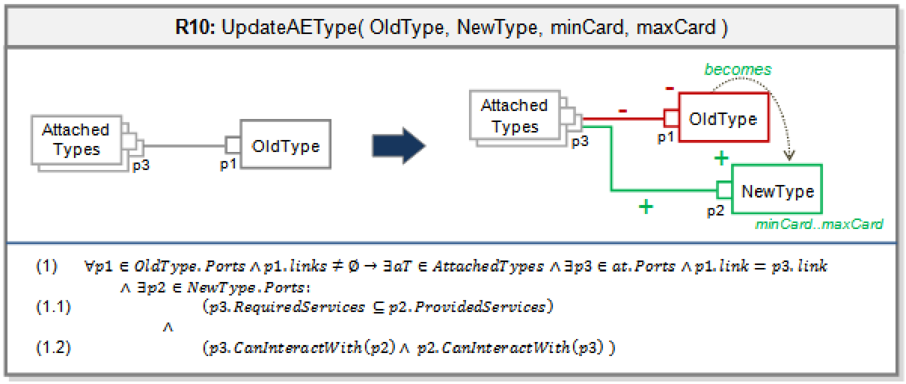
Since interactions among architectural types are performed through ports (i.e. they are the points of interaction), compatibility for updating is evaluated through ports. Thus, the requirement to perform an update operation is that the new type provides a set of ports syntactically and semantically compatible with the interacting ports of the old type. On the one hand, we define a port p_x as *syntactically compatible* with another port, p_y, if it provides all the services that are required from p_y, which are defined by the set of existing interactions of p_y, with exactly the same signature (i.e. name and parameters) of each service. Note that syntactic compatibility only refers to the minimum set of services *provided* by p_x: this means that a syntactically compatible port may provide additional services or require different services than the original. The goal of the updating operation is to guarantee that the updating does not break the current architecture, without taking care of the introduction of new required services. This is the responsability of other evolution rules, which will evaluate if all the required services are conveniently bound, thus guaranteeing the consistence of the architecture.

On the other hand, we define a port p_x as *semantically compatible* with another, p_y, if it provides the same observable behaviour as other elements expected from p_y. That is, the execution of the services of p_x must produce the same expected results, or sequence of traces, that p_y produces. However, the evaluation of semantic compatibility is challenging and its integration in the evolution model is not trivial. Since this is not the goal of this thesis, the reader can refer to other works (Engels et al., 2001) (Engels et al., 2002) in order to get more details about how some issues have been addressed from a formal perspective. For the sake of simplicity, we have abstracted semantic evaluation this way: each port is provided with a function, *CanInteractWith*, which evaluates if another port satisfies a certain contract or interaction protocol, i.e. that the observed behaviour is the required. For instance, in a port that requires compression and decompression services, a very simple function could evaluate that the compression of a sample data is decompressed correctly to the original data. That is, each port has the responsability of validating that their required services are provided correctly. Thus, we could say that a port p_x is *semantically compatible* with another, p_y, if all the ports that are connected to p_y, and request services from p_y, can interact seamlessly with p_x. In case of semantic incompatibility, a factible solution is

[45] In fact, we could remove unused functionality from a type by removing the old version and adding the new, reduced one. However, in this case, the state migration of its instances would not be performed.

the use of dynamically generated adaptors (Cámara et al., 2008) (Canal et al., 2008).

Syntactic and semantic compatibility is reflected in the *UpdateAEType* rule by means of condition (1) (see Figure 8.10): the execution of the rule is only performed if the ports of the new type version (i.e. parameter *NewType*) are syntactically (see condition 1.1) and semantically (see condition 1.2) compatible with the ports of the old type version (i.e. parameter *OldType*). The set of types that interact with *OldType* are modelled by means of a multiobject called *AttachedTypes*. The ports of the types that interact with *OldType* (i.e. variable *p3*) are used to evaluate the syntactic and semantic compatibility of the ports provided by *NewType* (i.e. variable *p2*). Moreover, since *AttachedTypes* includes all the types connected to *OldType*, in case *OldType* were connected to itself, *AttachedTypes* would include OldType in its set. This guarantees that the updating of a self-interacting type is made consistently: the ports of the new version must be semantically compatible with the ports of the old version. Self-interacting types are common in self-organised systems: different (distributed) instances of the same type (e.g. agents) interact themselves, sharing a common interpretation of the environment. In this case, semantic compatibility guarantees that instances of the old version can interact consistently with instances of the new version.



(1) $\forall p1 \in OldType.Ports \wedge p1.links \neq \emptyset \rightarrow \exists aT \in AttachedTypes \wedge \exists p3 \in at.Ports \wedge p1.link = p3.link$
$\wedge\ \exists p2 \in NewType.Ports:$

(1.1) $(p3.RequiredServices \subseteq p2.ProvidedServices)$

$\wedge$

(1.2) $(p3.CanInteractWith(p2) \wedge p2.CanInteractWith(p3))$

Figure 8.10. Rule R5 - UpdateAEType

The execution of this rule introduces the new type version tagged for addition (i.e. see the symbol '+' near the *NewType* component), and tags the old type version for removal (i.e. see the symbol '-' near the *OldType* component). The use of the addition and removal tags activates or constrains the behaviour of other instance-level rules. For instance: the rule *CreateArchitecturalElement* (see Figure 8.7) not only will create instances of the new type version, but will also avoid the creation of instances of the *old* type version. Another example: the rule *ActivateInstantiatedElements* (see Figure 8.8) will only allow the activation of instances of the new type version if and only if the minimum cardinality is satisfied.

However, in order to distinguish an update operation (which requires state migration) from an addition or removal operation (which also introduce addition/removal tags but do not preserve the instance state), this rule also introduces a relationship among the old type and the new type: the relationship “becomes”. It specifically models which type is going to replace the older version, and activates the rule *ReplaceAE* (which is described below) for performing the migration of the instances to the new type version.

With respect to interactions, the existing connections (i.e. links to *AttachedTypes*, a multiobject which represent the set of types that are interacting with *OldType*) are unlinked from the type to update (i.e. *OldType*) and linked to the new type version (i.e. *NewType*). This is modelled by tagging the old links with the symbol ‘-‘ (i.e. a removal tag) and the new ones with the symbol ‘+’ (i.e. an addition tag). These tags will avoid the creation of new attachments at the instance-level among instances of *OldType* and *AttachedTypes*, promoting instead the creation of attachments with instances of *NewType*. In this way, *OldType* will be progressively removed from the System.

♦ Rule R37: ReplaceAE

On the other hand, the *ReplaceAE* rule models the replacement of an instance by a new one, migrating its previous internal state and updating its existing connections (see R6, Figure 8.11).

This rule is activated if and only if a component instance is detected in a running System which matches the following conditions: (1) the type of such instance (which matches in the rule with *OldType*) has a “becomes” relationship with another type (i.e. *NewType* in the rule); and (2) such instance has reached a quiescent status (represented in the rule by the symbol “[ ]” around the instance *S1*). The first condition detects that the matching type has been updated. The second condition ensures that the interactions of the instance to migrate are stopped, and that the instance state is consistent, ready to be migrated. This is guaranteed by the quiescent status (Kramer & Magee, 1990), which is only achieved when there are no running and/or pending transactions.

The result of the execution of the *ReplaceAE* rule is the migration (or transformation) of an instance of the old type to an instance of the new type. This migration is modelled by means of the modification of the *instance_of* relationship and the transformation of the internal state. An instance of a type that has been updated (i.e. *S1* is an *instance_of* the type *OldType*, which will *becomes NewType*) is transformed to an instance of another type (i.e. in the right hand part of the rule R6, *S1* is now an *instance_of* the type *NewType*). This results in that the internal state of the instance (i.e. the *State* element inside the *S1* instance, in the left hand part of the rule R6) is transformed to another (i.e. *f(State)*, in the right hand part of R6), by means of a state migration function. This function is modelled as *f()* and defines the mappings from the old data structures to the new ones.

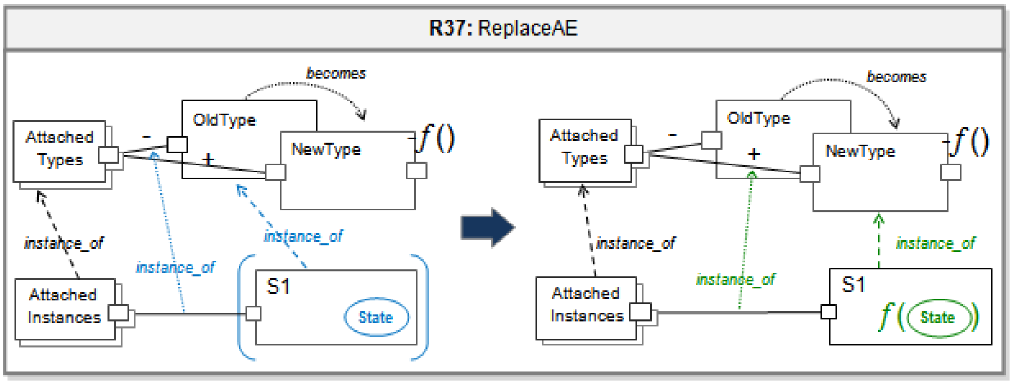


Figure 8.11. Rule R37 - ReplaceAE

There are two conditions to enable the state migration of instances: (i) the accessibility of their internal state, and (ii) the availability of a state migration function. On the one hand, if the old component type does not make accessible somehow the internal state of its instances, obviously we cannot migrate their state. For this reason, one of the conditions is that the old type provides a mechanism to get the internal state of its instances at runtime: this can be achieved by means of reflection, or by specialized functions that return the internal state to authorized requests (e.g. the migration function of a new version of the same type). This condition is explicitly included in the rule R6: the *State* of *S1* is visible, at least in the context of the rule. However, the rule does not reflect to *whom* it is accessible. It will depend on the specific implementation.

On the other hand, the new component type must provide a function to create new instances from the data structures of a previous type version. Otherwise, the state of an old instance could not be introduced into a new instance. The implementation of this function can be provided by means of a specialized constructor of the new type version which creates a new instance from the state of an instance of the previous type version. This condition is also explicitly included in the rule R6: *NewType* provides a function *f*, which results in the transformation of the original state of *S1* after the execution of the rule. The specific details for the creation of state migration functions are outside the scope of this paper. This rule only models the presence of such functions and what is the result obtained after the execution of the rule. If a state migration function is not provided (or the state of the instance is not accessible), then the old state cannot be migrated and is simply lost. However, the reader can refer to the works of (Ritzau & Andersson, 2000) or (Vandewoude & Berbers, 2005) for further details about how to automate the creation of state migration functions.

Another result of the execution of the *ReplaceAE* rule is the updating of the links of the instance to evolve. The set of instances that are interacting are represented by a multiobject called *AttachedInstances*, and their corresponding types by another multiobject, *AttachedTypes*. Note that, as a result of the execution of the type-level updating rule, the interacting types (i.e. *AttachedTypes*) are semantically compatible with the updated type (i.e. *NewType*). As a consequence, their instances (i.e. *AttachedInstances*) will be also semantically compatible (i.e. they could interact correctly) with the instance after evolution. The updating of links is modelled by means of the modification of *instance_of* relationships (which link the instance-level with the type-level): the attachments (i.e. links among instances) belong to different *attachment types* (i.e. patterns of interaction) before and after the execution of the rule. This means that, when the rule is executed, the existing links with other instances are deleted and replaced by new ones, but which point to the updated instance instead of the old one. Then, all the elements could start interacting.

Finally, note that the rule *ReplaceAE* is abstract enough to model two kinds of update approaches: *type substitution* and *type transformation*. One the one hand, in type substitution approaches, which has been commonly used in dynamic updating approaches (e.g. (Malabarba et al., 2000) (Ritzau & Andersson, 2000) (Segal & Frieder, 1993)), updating is performed through the *replacement* of instances of the old version by instances of the new version. Old instances are completely stopped and their state is migrated to new, updated instances. On the other hand, in type transformation approaches, updates are performed by the *transformation* of the internal structure of old type instances to accommodate the elements introduced by the new type. From outside, an evolved instance keeps the original boundaries, links and state, but also integrates the behaviour added by the new type. From inside, only the elements that have been affected by the change are modified: their state is migrated to new ones, whereas the state of non-changed elements is kept intact. Type transformation approaches are better suited to partially change large architectural types (e.g. servers), because they do not require stopping the entire architectural instance, but only the required parts of the instance.

This is the focus that has been used in our proposal to model recursively the evolution of PRISMA Systems. From outside, the evolution of a System is perceived as a type substitution: its instances are *replaced* by new versions having their state migrated. However, from inside the evolution is performed as a type transformation: the differences among the new type specification and the current specification are used to incrementally change the original instances, by means of a set of evolution operations. This strategy can be recursively applied, until (i) the decomposition of a type is not advisable (i.e. more than the 60% of type structures are going to be changed), or (ii) the internal composition is neither available or modifiable (e.g. COTS).

8.3.3.2 Graph-based Abstract Syntax

The architecture-based transformation rules presented in the previous section are formalised by mapping them to a graph-based abstract syntax. In this way, transformation rules can be introduced and executed in a graph transformation tool, such as AGG (AGG, 2010), to simulate and validate their execution. This will also allow us to analyze some properties and dependencies from a high abstraction level (e.g. interaction dependencies, instance-level implications, etc.).

To do this, the first step is the definition of a *type graph*, which describes the domain of a system and the terms that can be used to define the transformation rules.

Definition of the Type Graph

A *type graph* defines the concepts and relations of a specific domain: it represents a metamodel. Instances of this metamodel are called *instance graphs*. They represent the runtime states of a system and are subject to modification by transformation rules. Given a valid instance graph and a rule, the rule could be applied to the graph to produce a new graph. If this graph satisfies the constraints of the type graph (i.e. the metamodel), then it constitutes the successor state in the runtime model of the system. Note that elements of type graphs are not types in the architectural sense, but metamodel elements.

To model the evolution of both types and instances, an instance graph must include not only instances (e.g. composite instances: *Configurations*), but also the types that provide the behaviour of these instances (e.g. composite types: *Systems*). Therefore, a type graph must describe the ontological metamodel (Atkinson & Kuhne, 2003) of the concepts that are going to be subject to evolution. That is, the type graph must describe both the meta-types (i.e. the properties and relations among types) and the meta-instances (i.e. the properties and relations among instances). In addition, this type graph should also include the concepts required to describe how the concepts evolve over time: the *evolution tags*.

Figure 8.12 shows the type graph that includes the concepts of the PRISMA metamodel and the concepts related to the evolution of types and instances. The *ArchitecturalElement* concept represents simple architectural elements (i.e. not composites), which provide or request services through a set of *Ports* (see the *Port* concept in the metamodel). The *System* concept represents composite components: they are composed of several architectural elements (*AEType*), which are connected by means of attachments (*AttachmentType*) and/or bindings (*BindingType*). Since a System is also an architectural element (e.g. it has ports), it inherits its behaviour from the ArchitecturalElement concept. The *AEType* concept provides an alias for an architectural element that is imported into a System and the allowed cardinality in such System.

There are three kinds of evolution tags: *Added*, *Removed*, and *Update*. However, the latter is only used by stateful entities, i.e. architectural elements. The reason is that the update of stateless entities (like attachments, bindings and ports) can be reflected by means of a removal tag and an addition tag. In the case of stateful entities, this is not enough, since the state must be migrated from the old entity to the new one. For this reason, the *updated* concept has a link (called *replaced_by*) to the architectural element that is going to replace the tagged element. All the evolution tags (see the concept *Evol_Tag*) have a link (see the link *pending_evol*) to System instances (i.e. Configurations) that are pending to be evolved.

Finally, the metamodel also defines the elements of the instance-level, their properties and relationships (see Figure 8.12, meta-instances). This is needed because transformation rules will not only change types, but also instances. Since a metamodel defines the domain which is subject to changes, then it must also include the definition of instances. Note also how the *quiescent* status has been added to the concept *AEInstance*: this has been done this way to reflect when an instance is ready to be evolved.

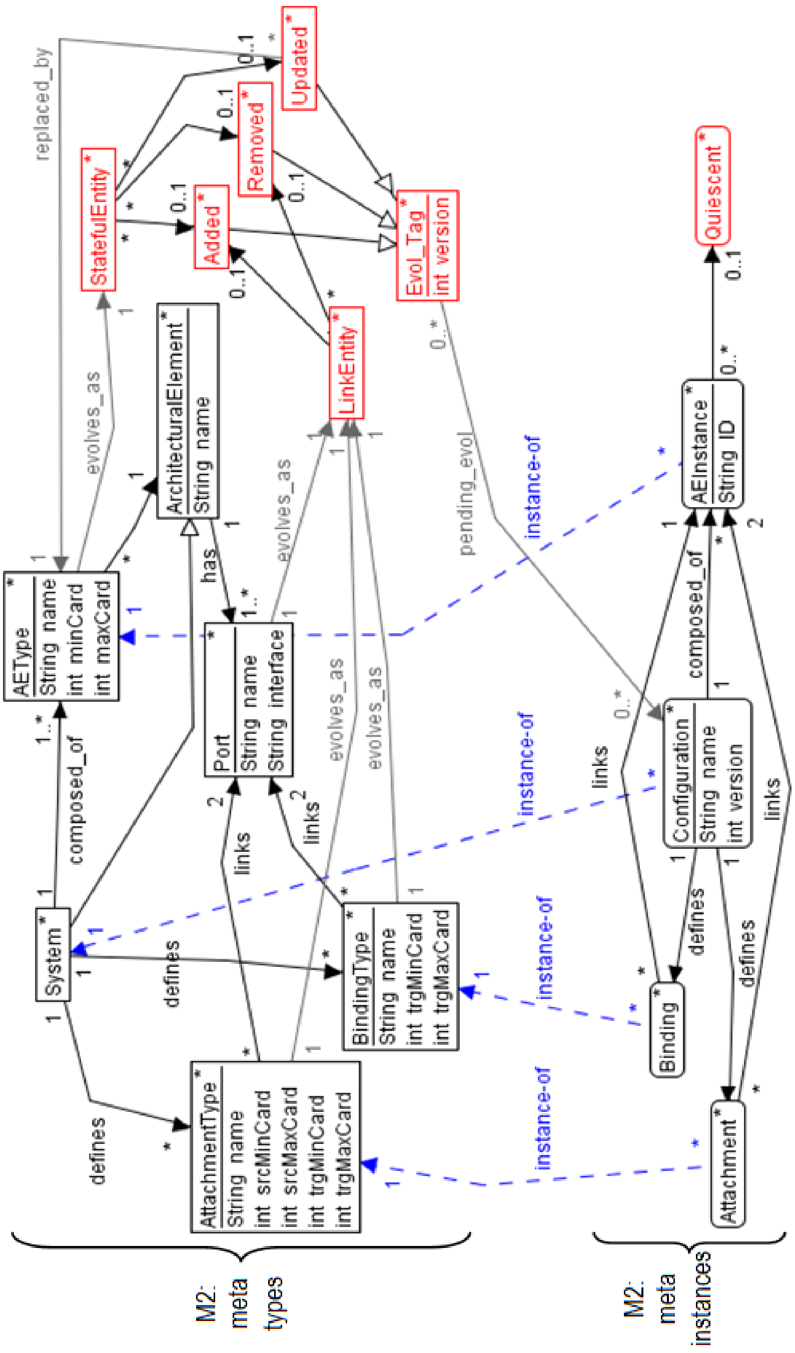


Figure 8.12. Type Graph of PRISMA with Evolution Tags

Figure 8.13 illustrates how an instance graph looks like: it includes type-level concepts and instance-level concepts. In particular, the type-level concepts included in this graph are related to the System *Sys*, after the type-level execution of the evolution process "E_0:*Sys*.v_0→*Sys*.v_1": it removes the architectural type *B* and adds the type *C* (see Figure 8.2, page 324). The System *Sys* is modelled as a graph, and each one of its structural elements (ports, architectural types, and connections) are modelled as nodes of this graph, according to the type graph described above. For instance, the type *A* is modelled with two graph nodes: (i) an *ArchitecturalElement* node named "Comp_A" which defines the behaviour of the type and which is linked to the *Port* nodes named "PClient" and "PServ"; and (ii) an AEType node named "A" which defines the usage restrictions of such type in *Sys*: a minimum and maximum cardinality of 1. Note how the elements that have been affected by the evolution process (i.e. the attachment types *Att_AB*, *Att_AC*, and the architectural elements *B*, *C*) have been tagged with removal or addition tags. Finally, the instance graph shows some elements of the instance-level (see M0): the Configuration *C1*, which is still conforming to version 0 of *Sys*.

This graph is a clear example of the benefits of using an architecture-based concrete description instead of a graph-based abstract description to describe the evolution semantics. An architecture-based description of a system (e.g. see Figure 8.2, page 324) is more concise than a graph-based abstract description. However, the former cannot be automatically validated and/or formally analysed to detect inconsistencies.

Once the type graph describing the metamodel subject to evolution has been defined, then graph transformations can be defined. Next it is shown how an architecture-based evolution rule, *CreateArchitecturalElement*, has been mapped to a graph-based abstract description.

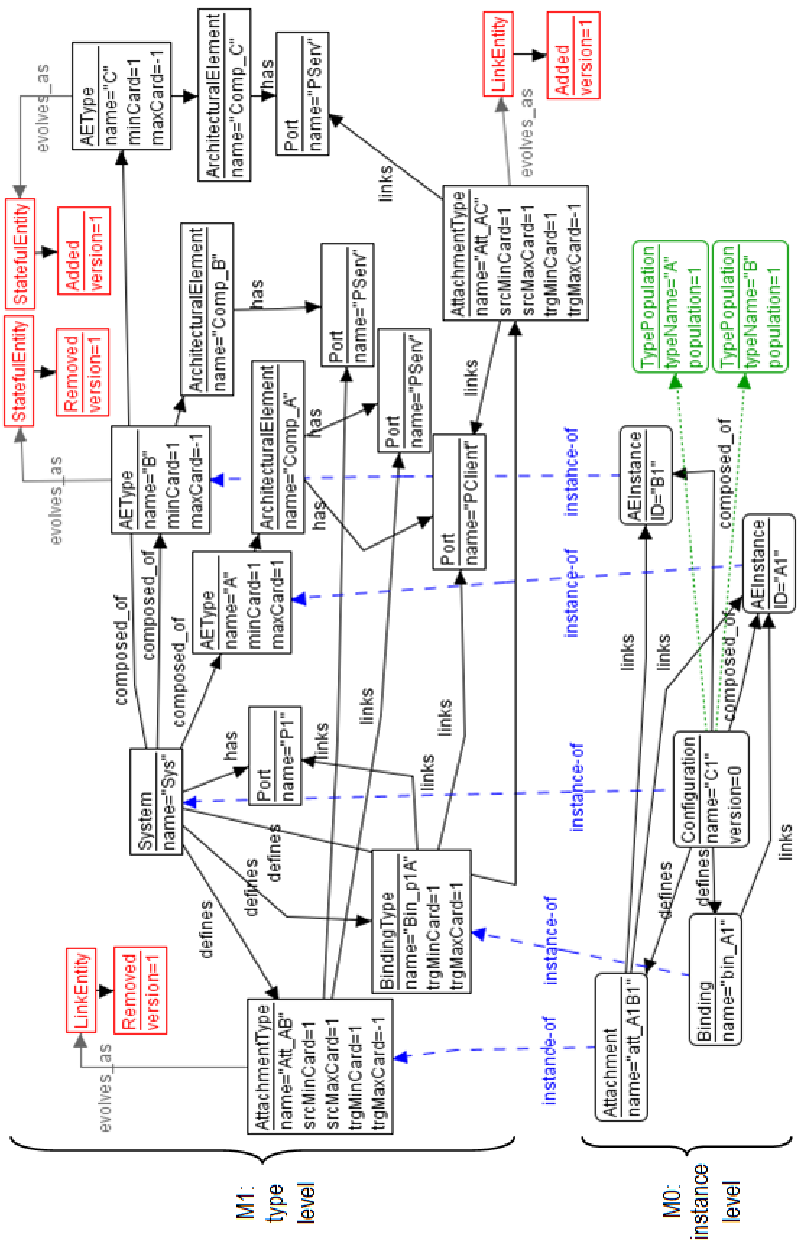

AEType name="C" minCard=1 maxCard=-1
ArchitecturalElement name="Comp_C"
Port name="PServ"
LinkEntity
Added version=1
StatefulEntity
Added version=1
StatefulEntity
Removed version=1
evolves_as
ArchitecturalElement name="Comp_B"
Port name="PServ"
AttachmentType name="Att_AC" srcMinCard=1 srcMaxCard=1 trgMinCard=1 trgMaxCard=-1
TypePopulation typeName="A" population=1
TypePopulation typeName="B" population=1
AEType name="B" minCard=1 maxCard=-1
ArchitecturalElement name="Comp_A"
Port name="PServ"
Port name="PClient"
AEInstance ID="B1"
AEType name="A" minCard=1 maxCard=1
AEInstance ID="A1"
System name="Sys"
Port name="P1"
Configuration name="C1" version=0
BindingType name="Bin_p1A" trgMinCard=1 trgMaxCard=1
LinkEntity
Removed version=1
AttachmentType name="Att_AB" srcMinCard=1 srcMaxCard=1 trgMinCard=1 trgMaxCard=-1
Attachment name="att_A1B1"
Binding name="bin_A1"
has
links
composed_of
defines
instance-of
M1: type level
M0: instance level

Figure 8.13. Instance graph of the Sys type (at version 1) with the C1 instance (at version 0)

Mapping of an evolution rule: CreateAE

An example of the mapping of an evolution rule to a graph transformation rule is shown in Figure 8.14. This figure shows how the evolution rule *CreateArchitecturalElement* (see Figure 8.7, page 332) has been modelled in AGG. The execution of this rule (which has been renamed as *CreateAE*) requires three input parameters: *configName*, the name of the Configuration where a new instance is going to be created; *typeToInstantiate*, the name of the type to instantiate; and *newID*, the identifier of the new instance to create.

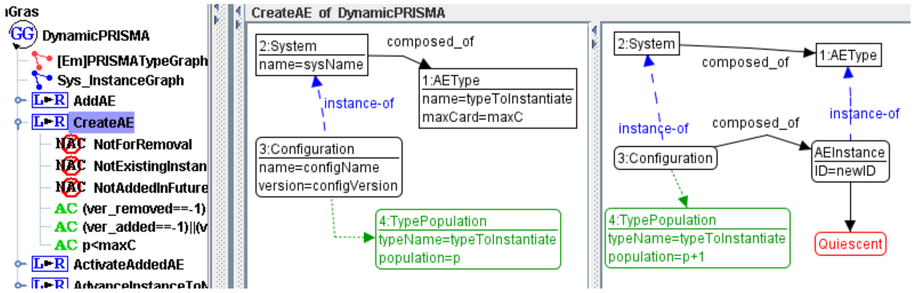


Figure 8.14. CreateAE: mapping of rule CreateAE in the AGG tool

The left-hand side of the rule describes the initial matching (see Figure 8.14, center) required to execute the rule. The instance graph must contain a Configuration node whose attribute "name" equals to *configName*. This node must be linked to a System node (by an *instance-of* relationship), which in turn is linked to an AEType node with an attribute "name"=*typeTo Instantiate*. This checks that the type to instantiate is declared in the Configuration type (i.e. the System node *sysName* linked to the Configuration). The right-hand side of the rule describes the result of the transformation rule: a new AEInstance node, whose attribute "ID" is *newID*, has been created and linked to the selected Configuration node (i.e. *2:Configuration*). In order to reflect the fact that the new instance is in a quiescent status, it is also linked to a Quiescent node.

The complex conditions defined by the rule *CreateArchitecturalElement* (see Figure 8.7) have been modelled by a set of *NACs* (Negative Application Conditions) and attribute conditions (see Figure 8.14, left). These conditions allow the execution of a graph transformation rule only if: (i) none of the defined NACs matches with the instance graph, and (ii) all of the attribute conditions are true. Here is described how these conditions have been modelled in AGG.

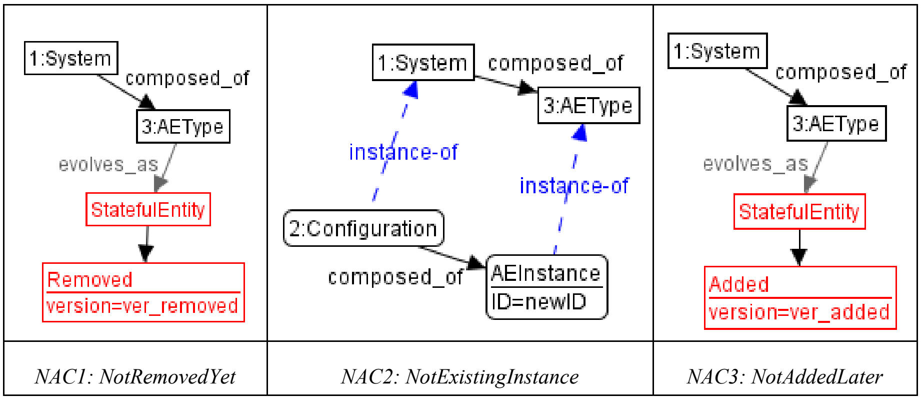


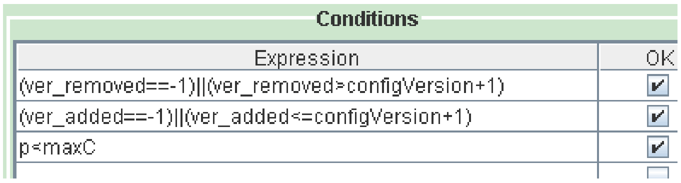
Figure 8.15. NACs of the graph transformation rule *CreateAE*

(i) If *typeToInstantiate* has been removed from the System type (i.e. it has been tagged for deletion), it must have been removed in a System version greater than the current Configuration version. This is checked by a NAC and an attribute condition. NAC1 (see Figure 8.15) checks if the type to instantiate (i.e. the node *3:AEType*) is tagged for deletion (i.e. it is linked to a Removed node). If true, the variable *var_removed* will contain this tag version number; otherwise (the type has not been deleted yet), it will contain the value -1. This variable is compared with the current Configuration version (see the first attribute condition in Figure 8.16), checking that it is greater. If it is evaluated to false, then the rule *CreateAE* cannot be executed.

(ii) The identifier of the new instance, *newID*, must not have been used previously. This is checked by NAC2 (see Figure 8.15): if the selected configuration (i.e. the node *2:Configuration*) is linked to an AEInstance node with an ID=*newID*, then the NAC condition is true and the rule *CreateAE* cannot be executed.

(iii) If *typeToInstantiate* has been added at runtime (i.e. it is tagged with an addition tag), it must have been added in a System version less than the current Configuration version. This is similarly checked as with the removal case. See NAC3 in Figure 8.15 and the second attribute condition in Figure 8.16.

(iv) The number of existing *typeToInstantiate* instances in the Configuration must be lower than the maximum cardinality defined in the System type. This is checked with the help of an auxiliary node, TypePopulation (see Figure 8.14), which keeps the number of instances (attribute *population*) of a type (attribute *typeName*) that have been instantiated in the Configuration it is linked to. Then, an attribute condition (see the third attribute condition in Figure 8.16) checks that the population, *p*, of the selected TypePopulation node (i.e. this which attribute *typeName* equals to the type to instantiate, *typeToInstantiate*) is less than the maximum cardinality, *max*C, of the selected *AEType* node.


Conditions

Expression	OK
(ver_removed==-1)||(ver_removed>configVersion+1)	✓
(ver_added==-1)||(ver_added<=configVersion+1)	✓
p<maxC	✓

Figure 8.16. Attribute conditions of the graph transformation rule *CreateAE*

In a similar way, the other architecture-based evolution rules are mapped to graph-based transformation rules. However, due to the high number of evolution rules involved, only the most representative ones have been mapped to graph-based rules. The simulation and analysis of the graph-based transformation rules has been left as a future work.

8.3.4 Discussion

This approach captures the dynamic evolution process at a very high level of abstraction, as a sequence of gradual, asynchronous changes on both the type and instance levels. Typed graph transformations have been chosen to describe the evolution semantics because they naturally model both the system itself (as a graph of types and configurations) and the asynchronous nature of its evolution (by the individual application of rules without global control). As opposed to logic-based formalisations or process calculi approaches (Canal et al., 1999), (Cuesta et al., 2004), the use of graphs allows us to represent the system and its runtime state at a high level of abstraction. The direct mapping between graphs and visual models resembles that between UML diagrams and their metamodel-based abstract representations. It has been shown that graph transformations allow us to describe precisely and concisely the principles and mechanisms of dynamic evolution: (i) how, in presence of change, will the involved type and instances react, and (ii) how will their interactions with other elements be managed.

However, the details of how the quiescence status is achieved by instances have been explicitly excluded from the graph-based model. The reason is that this would require modelling also the instance execution model: to describe the safe stopping of running transactions and the blocking of new service requests, the model should also include how instances process and execute service requests. This would add excessive complexity on the evolution model, thus eliminating the benefits of a concise description. We have decided to simply model quiescence as an attribute of an instance, which describes when this status is reached. The concrete semantics have been left to the infrastructure (i.e. the middleware).

Another aspect that has been omitted from the transformation rules is the evaluation of semantic compatibility among ports. It has been modelled in the rules as a function, but this function has not been detailed. The implementation of mechanisms to establish compatibility among types has been left as a future work.

As a result of introducing the architecture-based transformation rules in AGG, some limitations have been found concerning the use of advanced concepts, such as complex application conditions or multi-objects. These advanced concepts have been used to describe complex rule preconditions (e.g. that the cardinality of a given architectural type is preserved when creating new instances) or to describe operations on sets of nodes (e.g. quiescing all the instances of a given type, or preserving the existing interactions of a given instance). However, the modelling of these advanced concepts is still not supported in AGG, so different ways of overcoming these limitations have been addressed (e.g. by using the Java interface provided by AGG or by assuming that a single node models a multi-object). The limitations found are the following: (i) complex application conditions with nested *if-then-else* conditions or that operate on sets of nodes, cannot be defined (this must be done in Java through the use of the internal API); (ii) multi-objects cannot be modelled, rules can only operate on a set of nodes which their exact number is known in advance (we cannot define universally quantified operations); (iii) rules that invoke other rules and provides some parameters, cannot be directly modelled (it requires the usage of auxiliary nodes to pass information among rules); and (iv) the definition of composite rules, which would allow the reuse of rules and the management of different levels of abstraction, is not currently supported.

8.4 Conclusions & further works

This chapter has introduced the semantics of the asynchronous evolution of types in the context of software architecture. This is an important feature for the design and development of large systems with a long-time usage, since it allows the introduction of concurrent changes at runtime without waiting to sequence all changes, which would delay the introduction of needed capabilities or problem fixes. This section summarizes the contributions of this chapter, the remaining works, and the related publications.

8.4.1 Conclusions

This work has been focused on the dynamic evolution of the structural view of systems (i.e. the architecture of their composites). This has been addressed from a white-box perspective: the internal structure of composite instances is evolved gradually, by adding or removing its elements at runtime, connecting/disconnecting them, etc. From a black box perspective, this is perceived as a replacement of the entire architecture and the migration of their state. However, internally only some parts have been changed, while keeping the state of the other elements.

The evolution semantics described allows the concurrent, but ordered, dynamic evolution of both the type and its instances. The evolution is concurrent because both the type and its instances evolve asynchronously, independently of each other. But the evolution is also ordered because instances evolve towards the last updated version of the type, while preserving the order in which different evolution processes (i.e. those that create new type versions) were introduced. Version management is carried out by means of evolution tags, which allow keeping evolution traceability, so that each instance can follow the evolution of its type across the time. Thus, although a type would evolve several times, at the end each instance would converge to the last version of the evolved type.

8.4.2 Further works

Substantial research remains to be done, such as the definition of a fully distributed and decentralized asynchronous evolution model. The model presented is decentralized at the instance-level. Each instance evolves at different times, without require being located at the same node where the evolution started. However, the model relies on a centralized type which receives the evolution requests and propagates the changes to its (distributed) instances. In a fully decentralized model, a type may also be distributed among different nodes. In this case, when an evolutionary change is requested on a type, it should be propagated properly to the other nodes. Then, some issues arise like: (i) how to keep (distributed) type specifications synchronized; (ii) how to manage concurrent type evolution requests and avoid type version branching. These issues are similar to those dealt in versioning management approaches (De Lucia et al., 2009), (Thao et al., 2008).

In addition, there are issues that have not been included in the transformation rules, such as the management of runtime faults. Since rules only describe preconditions and postconditions of actions, they cannot model what happens in the middle of such actions. For instance, if an evolution operation cannot be finished for any reason, then the changes must be undone. The transactional management of evolutions have been considered in the implementation of the evolution infrastructure (see page 306), but not in the graph-based modelling. This has been left as a future work.

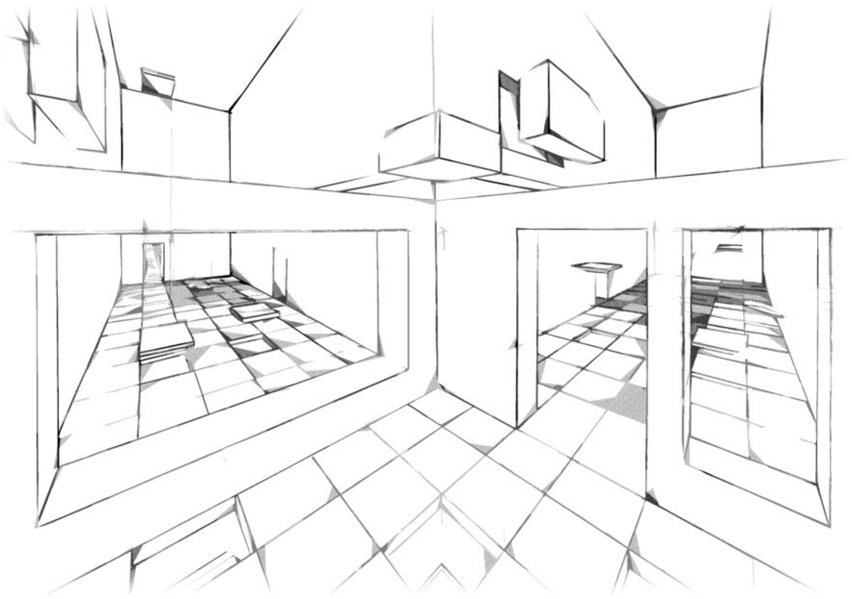
8.4.3 Results

The work related to the description of the asynchronous evolution semantics of architectural types has been presented in the following papers:

- **Costa-Soria C.**, Heckel R. *Modelling the Asynchronous Dynamic Evolution of Architectural Types*. In Weyns, D.; Malek, S.; De Lemos, R.; Andersson, J. (eds.): Self-Organizing Architectures *(Revised and extended papers)*. Lecture Notes on Computer Science, vol. 6090, pp. 198-229. Springer-Verlag, Berlin Heidelberg, July 2010.
- **Costa-Soria C.,** Heckel R. *Formalizing the Asynchronous Evolution of Architecture Patterns*. In proc. of: Workshop on Self-Organizing Architectures (SOAR'09), held at the Working IEEE/IFIP Conference on Software Architecture (WICSA'09) and European Conference on Software Architecture (ECSA'09). Cambridge, UK, 14th September 2009.

PART IV

CONCLUSIONS & FURTHER WORK



CHAPTER IX

CONCLUSIONS

This chapter presents a summary of the contributions of the thesis, the evaluation of the approach, and the evaluation of the research method in section 9.1. The chapter also presents the results of the thesis in section 9.2. This chapter also presents future work that can be done to continue this research in section 9.3.

9.1 Conclusions

A software system, during its lifetime, may require several updates, improvements, or new features. If these change requirements are not conveniently addressed, then the risk that the software system ages prematurely increases (Parnas, 1994). As a result of this aging, the system may become useless, due to its inability to meet the functionality, performance or stability needs of its users. For this reason, from the early conception of a software system, software evolution must be borne in mind. Software evolution acquires even more importance in the case of safety- and mission-critical software systems, which cannot be stopped to perform maintenance or evolution operations due to their continuous operation. To reduce the aging of these critical systems, they must be provided with mechanisms enabling their *dynamic evolution*, i.e. supporting changes while they remain in operation.

Dynamic Evolution is a feature which, despite having acquired a lot of attention during the last decade, is still interesting to the research community. This has been motivated by the increasing scope and requirements of dynamic evolution: from the initial objective of changing a single procedure at runtime, to the long-term goal of building complete self-managed software systems.

This section is structured as follows. Section 9.1.1 summarizes the contributions of this work. Next, section 9.1.2 presents the evaluation of the approach, according to the criteria presented in Chapter 4. Finally, section 9.1.3 evaluates to which extent the goals of the thesis have been satisfied and how the research has been conducted.

9.1.1 Contributions

This thesis takes another step forward in the development of dynamic evolvable systems. Starting from existing results (e.g. safe stopping strategies, state migration, code generation, reflective models, etc.), this thesis has provided new contributions to the area. We would like to emphasize the following contributions: (i) the combination of dynamic reconfiguration and type evolution, (ii) the encapsulation of evolution and reconfiguration concerns into aspects, and (iii) a model for autonomic reconfiguration and asynchronous evolution of types. These contributions, and other results of the thesis, are briefly described next.

A dynamic architecture-based approach

This thesis presents a framework to build architecture-based, dynamically evolvable, software systems. The fact that this framework is an architecture-based approach provides the following advantages: (i) it offers a high-level of abstraction for describing dynamic changes; (ii) it allows varying the level of system description; and (iii) it advantages from the existing support provided by architecture description languages, such as modelling, code-generation, and formal analysis. This framework uses generic architectural terms so that it can be applied to any architecture description language.

In particular, this framework has been applied to the PRISMA architecture description language, a language that does not support the description of software systems with dynamic evolution requirements, but that provides support for: (i) the structural and behavioural modelling of aspect-oriented software architectures and, (ii) the automatic generation of executable code. The result is *Dynamic PRISMA*, an extension of the PRISMA model which provides dynamic evolution support.

Integration of dynamic reconfiguration and dynamic type evolution

The main contribution of Dynamic PRISMA is the combination of two levels of dynamism: *Dynamic Reconfiguration*, addressing changes at the configuration level (i.e. the architectural configuration), and *Dynamic Type Evolution*, addressing changes at the type-level (i.e. the specification of architectural types and instances). Therefore, in Dynamic PRISMA a software system is not only able to reconfigure at runtime the building blocks it is composed of (i.e. architectural types), but also to redefine these building blocks (or introduce new ones) at runtime. This provides software systems with a high level of plasticity and flexibility.

Plasticity is provided by means of dynamic reconfiguration. In Dynamic PRISMA, this is related to changes on the structure of a specific composite instance[46] maintaining the conformance to its composite type pattern. Thus, a composite instance has a degree of plasticity in the sense that it tolerates changes on its structure without breaking the design decisions defined by its composite type.

Flexibility is provided by means of dynamic type evolution, which in Dynamic PRISMA concerns to changes on the specification of an architectural type (either simple or composite) and the propagation to its instances. Thus, an architectural type is flexible in the sense that it can be redefined at runtime to meet new, unforeseen requirements.

A model for autonomic reconfiguration and asynchronous type evolution

Another contribution of the thesis is how the two levels of dynamism have been supported in the framework. On the one hand, dynamic reconfiguration is supported by means of *autonomic capabilities*, which enable each composite instance to reconfigure its structure at runtime according to either internal or external stimuli. This reconfiguration can be either reactive (i.e. externally-driven) or proactive (i.e. internally-driven). On the other hand, dynamic type evolution is supported by means of *asynchronous reflection*, which enables a system to change the definition of its architectural types at runtime (both their specification and executable code), while allowing each type instantiation to delay its transformation until it is ready for evolution.

Both levels of dynamism are provided with *consistency management support* to preserve the integrity of the system before and after a dynamic change. This support is provided through the integration of: (i) safe stopping mechanisms, which guarantee that the artefacts to be evolved are placed in a safe state; (ii) state transfer mechanisms, which allow software artefacts to preserve their state among updatings; and (iii) transactional support, which allows a system to revert dynamic changes if anything fails.

[46] Note that dynamic reconfiguration only concerns to changes on *composite elements*. Simple elements do not describe architectures and because of that, their change is expressed as type evolutions, not dynamic reconfigurations.

Encapsulation of evolution and reconfiguration concerns into aspects

A third contribution of this thesis is the identification of the concerns related to evolution and their integration in the framework through aspects. This improves the separation of concerns and allows us to change proactive reconfiguration specifications, evolution mechanisms, or the business logic independently of each other. This separation is possible because the dependencies among concerns (i.e. weavings) are made explicit, separately managed, and thus easy to be found in the system specification.

Validation of the evolution semantics in .NET

Dynamic PRISMA has been validated through its implementation in .NET to allow its integration with the existing tools of PRISMA, which are also implemented in .NET. The PRISMANET middleware has been extended to support the dynamic reconfiguration and evolution of architectural types and instances. In addition, the execution model of component instances has been modified to integrate the supporting mechanisms for safe stopping, state transfer, and transactional evolution. All the management of this functionality has been encapsulated into aspects, according to the framework.

Thus, Dynamic PRISMA can be integrated in the Model-Driven Development process of PRISMA: evolvable PRISMA architectures can be defined through the PRISMA Case Tool, their code can be automatically generated, and they can be directly executed on the PRISMANET middleware, which supports the dynamic evolution and reconfiguration of such architectures.

However, only the most important details of the implementation (e.g. the reification of types, the generation of new code, the updating of instances, etc.) have been included in this thesis for space reasons. Further implementation details can be found in the Ms. Science thesis that have been developed in the context of this thesis and that have been supervised by the author: (Aliaga-Varea, 2008), (Hervás-Muñoz, 2009).

Validation of the evolution semantics through graph transformations

However, since the executable code cannot be directly used to disseminate the results across the research community and may be still subject to unforeseen failures (because it does not support precise analysis and validation), Dynamic PRISMA has been also validated by means of a high-level formalism. Specifically, the semantics of evolution has been precisely described by means of typed graph transformations.

This formalism has been chosen because it naturally models both, the system architecture and the asynchronous nature of its evolution. This has been realized by describing the observed behaviour of evolution services (both at the type-level and at the instance-level), and by using an architecture-based concrete syntax, which is more concise and easier to understand than the graph-based abstract syntax. This has facilitated the identification and resolution of some issues related to asynchronous evolution, such as the management of type conformance, the order of evolution processes or the coherence of interactions.

Application to the domain of autonomous robotics

All the contributions of this thesis have been illustrated through a case study from the domain of autonomous robotics, an area which could potentially benefit from the results of this thesis. In particular, an autonomous agricultural robot for plague control, called Agrobot, has been presented. The software architecture of this robot has been specified in PRISMA, focusing on the vision system and its dynamic evolution requirements. To achieve these evolution requirements, Dynamic PRISMA has been used to provide the ability to introduce new behaviour at runtime (by replacing aspects through type evolutions) and reconfiguring the structure at runtime (by changing components through dynamic reconfigurations). This has provided the Agrobot with tools to deal with changing environments at runtime, both foreseen and unforeseen. This could be promising for the design and development of the robots of the future.

9.1.2 Evaluation of the approach

This section presents a qualitative evaluation of the proposed approach, Dynamic PRISMA, and how it compares to other approaches from the literature. This evaluation is performed by means of the list of criteria used in Chapter 4 for comparing related works: *degree of formality*, *level of dynamism*, *activeness*, *separation of concerns*, *evolution management*, *introspection support*, *types of change*, and *consistency management*.

Degree of formality

Dynamic PRISMA defines a formal architecture description language (i.e. an ADL) supporting dynamic reconfiguration and type evolution. It is defined as an extension of the PRISMA ADL (Pérez, 2006), a formal ADL[47] that does

[47] The PRISMA Aspect-Oriented Architecte Description Language (AOADL) is based on a modal logic of actions for describing services (Stirling, 1992), and π-calculus with

not support the specification of evolvable systems but which allows the precise description of both the structure and behaviour of architectural models. As a result, Dynamic PRISMA increases the expressiveness of the PRISMA language, enabling the precise description of dynamic, reconfigurable, and evolvable, architectural models.

The semantics of dynamic reconfiguration and type evolution has been entirely specified by using the PRISMA ADL language, thus achieving a high level of precision in describing the operation of the underlying reconfiguration and type evolution mechanisms. In addition, the semantics of asynchronous type evolution has been described in terms of typed graph transformations, motivated by their expressiveness to describe how models evolve over time and their formal backgrounds. The reason behind this formalisation, both in PRISMA ADL and in typed graph transformations, is twofold. First, this formalisation allows appreciating the complexity of the processes without being lost in the details, and helps disseminating the semantics of these evolution processes. Second, this formalisation allows the analysis and verification of the properties of evolution processes, such as liveness (e.g. a component will always reach a quiescent state, an evolution transaction always ends) or safety (e.g. only quiesced components are changed, a component cannot be removed if another is still interacting with it). However, the analysis and verification of the semantics of evolution has been left as a future work.

The formal basis of Dynamic PRISMA advantages non-formal approaches in that it enables the automatic generation of code after evolutions, the realization of behavioural analyses (e.g. to automatically generate state transfer functions or to evaluate semantic compatibility among types on replacements), or the verification of system properties regarding changes (e.g. that a certain component will not be removed anytime due to reconfigurations, or that a type may be instantiated in the system lifetime). These features are manually done in non-formal approaches, being complex to perform (e.g. verification of system properties). The advantage of using a specification language with a formal background is that the structure and behaviour of the software system are precisely described, so automatic analyses can be performed to alleviate the architect or developer from some tasks. These advanced features have been left as potential future work (see sections 9.3.1 and 9.3.5 later in this chapter).

priorities for describing interactions among services (Milner, 1993). See section 2.4 (page 49) for more details.

Level of dynamism

This attribute refers to the kind of changes that can be done in a running system: *dynamic reconfiguration*, changes in the organisation of the building blocks of a system; and *dynamic type evolution*, changes in the structural and behavioural specifications of the building blocks of a system. Most of existing works have only considered one degree of dynamism, but not both: either dynamic reconfiguration in the area of Software Architectures, or dynamic type updating in the area of Component-Based Software Development.

Dynamic PRISMA features the integration of both levels of dynamism. This is done by following an autonomic approach to support changes at the instance-level (both for reconfigurations and instance-level type evolutions), and a reflective approach for communicating with the type-level. Dynamic PRISMA advantages other works in the way it combines both kinds of changes. It addresses architecture reconfiguration and architectural type evolution in a great detail, covering not only the specification of changes but also the underlying mechanisms involved. Regarding the area of self-managed systems, Dynamic PRISMA is the first that combines reconfiguration and type evolution.

Activeness

This attribute refers to the way that dynamic changes can be initiated or driven: *reactively* (externally introduced at runtime), *proactively* (internally initiated by the system), or *both*.

Dynamic PRISMA supports both kinds of activeness, for both dynamic reconfiguration and dynamic type evolution. This has been done through the definition of a common set of lower-level services supporting dynamic change (i.e. those encapsulated in the *Monitoring* and *Effector* aspects) and their coordination at a more abstract level through the different *Planning* aspects. Thus, reactive and proactive changes are driven by means of these common interfaces, although in different ways: *ports* for reactive changes, and the *Analysis* aspect for proactive changes.

This brings Dynamic PRISMA with expressiveness to build self-managed systems (which make extensive use of proactive changes) that are flexible enough to be extended with unanticipated changes at runtime (by means of reactive changes). Regarding the current state-of-the-art, most approaches provide support for both proactive and reactive changes, but *only* for dynamic reconfiguration. Dynamic PRISMA contributes to the area by supporting the the description of both proactive and reactive changes for both dynamic reconfiguration and type evolution. As further work, it remains the support for proactive non-programmed evolutions.

Separation of evolution concerns

This refers to whether the evolution concerns are explicitly separated from other concerns or not. And, in the case of proactive systems, whether evolution specifications are separated from the supporting mechanisms or not.

This has been one of the main goals behind the research concerning Dynamic PRISMA: the identification and isolation of the evolution concerns. As a result of this research, the concerns related to reconfiguration and type evolution have been identified and decomposed into several aspects, addressing diferent facets of the evolution process at the type-level and the instance-level: an aspect for introspecting the type, another dealing with the monitoring of an instance, another dealing with the generation of the evolved type, another dealing with the transformation of an instance, etc.

This separation helps to understand the complex processes that are behind dynamic change, and to identify (and reduce) the dependencies among the different processes. As a result, this helped in the implementation of the executable infrastructure as well as in the maintenance of the change specifications. In the literature, any work has addressed this level of decomposition of concerns in the description of dynamic changes.

Evolution management

With respect to the nature of evolution management (i.e. centralized or distributed), Dynamic PRISMA follows a hybrid approach, which varies depending on the kind of dynamism (i.e. reconfiguration or type evolution).

In case of dynamic type evolution, change requests can be provided by any element of the system. In this sense, the approach is distributed, because any software artefact can act as an evolving agent. These change requests are provided to the target type by means of reflection mechanisms, which are available from any point of the system. In addition, since type evolutions are performed by each type independently of the others, and by their respective instances, the management of type evolutions are also distributed.

In case of dynamic reconfiguration, change requests can be generated by the Evolver component of a composite component (i.e. proactive reconfigurations), or received from external sources (i.e. reactive reconfigurations). In both cases, the management of reconfigurations are performed by the Evolver of each composite component. In this sense, the approach is hierarchically distributed: each composite component reconfigures itself proactively and may induce changes in its internal components by means of reactive changes.

The distributed management achieved by Dynamic PRISMA aligns with the growing interest on the scalability issues of self-managed systems. However, due to space reasons, the distributed management of type evolutions and reconfigurations have not been described in detail in this thesis. This has been left to be covered in future publications.

Introspection

This attribute describes the degree of self-awareness that a system is provided with. With the aim of supporting proactive changes (both programmed and non-programmed), Dynamic PRISMA has been developed taking into account that an evolvable system must be aware of its structure and state.

Dynamic PRISMA provides different levels of introspection: at the architectural level, which is useful for proactive dynamic reconfigurations, and at the type level, which is useful for proactive dynamic type evolutions. At the architectural level, each composite instance can be aware of its internal composition (in terms of components, connectors and attachments), its provided and required services (in terms of ports and bindings), and its internal status. At the type level, each type also provides introspection services to obtain more details about its specification, which varies depending it is a composite type or a simple type. The difference among introspection at the type level and introspection at the architectural level is that the latter provides state information.

In comparison to state-of-the-art approaches, the introspection facilities provided by Dynamic PRISMA advantage in that they are type-oriented rather than system-oriented. Introspection is not provided to all the types of the system, but only to those types which are evolvable. And, since introspection is provided through service interfaces, the architect can customize which kind of introspection services will be available or not, to avoid exposing outside the internal structure and state of certain elements of the architecture. This is another contribution of this work that has not been considered in the literature until now.

Types of change

This attribute refers to the kinds of operations that can be performed to change software artefacts at runtime. In the literature, five operations have been identified: additions, removals, updates, linkings and unlinkings.

Dynamic PRISMA supports the five evolution operations. However, since Dynamic PRISMA operates at two levels of dynamism (i.e. dynamic reconfiguration and type evolution), these evolution operations are actually 10 in total: 5 that apply at the configuration level, and 5 that apply at the type level. Depending on the context of execution, these evolution operations have a different meaning: (1) *additions*, to add at runtime new architectural instances (by means of dynamic reconfiguration), or new types (by means of dynamic type evolution); (2) *removals*, to remove at runtime architectural instances (at the configuration level), types or parts of a type (both at the type level, by means of type evolution); (3) *updates*, to replace architectural instances or update types (or parts of a type); (4) *linkings*, to create links among architectural instances (by means of dynamic reconfiguration), communication patterns among architectural types, or relationships among the parts of a type (both by means of dynamic type evolution); and (5) *unlinkings*, to remove links, communication patterns or relationships.

In addition, a novelty of Dynamic PRISMA is that it introduces an implicit new kind of evolution operation: the *exclusion* operation. For each architectural type, both reconfiguration operations and type evolution operations can be customized by the architect, by selecting which evolution operations are allowed or not. This kind of operation, the *exclusion* of evolution operations, increases the expressiveness of the language. It is another contribution of this work that has not been considered in the literature (as an evolution operation) until now.

Consistency management

This attribute evaluates the presence of mechanisms, or strategies, for preserving the consistency of a system before and after a dynamic change. Three mechanisms have been considered: state transfer, safe stopping and transactional support. Given the importance of these mechanisms for preserving system consistency when dealing with dynamic change, the three mechanisms have been integrated in the semantics of Dynamic PRISMA.

Support for state transfer has been integrated in a transparent way in the semantics of reconfiguration and type evolution. The update/replace evolution operations apply state transfer whenever is possible: that is, if the source and target types provide the state transfer functions required for migrating the state (see page 217). Currently a *delegated state transfer* approach (see page 81) has been followed: the developer must define the state transfer functions. The automatic generation of these functions is feasible, but has been left as a future work (see section 9.3.5).

Support for safe stopping and *transactional execution* has been integrated as part of the middleware, since they rely on low-level structures, such as the management of service requests, state contexts, and interactions. However, the use of such mechanisms has been made explicit in the semantics to correctly analyse in which specific situations they are required and how they may impact other concurrent processes of the system.

The explicit integration of state transfer, safe stopping and transactional change in the semantics of dynamic reconfiguration and type evolution is another of the contributions of Dynamic PRISMA. In the literature, very few works have integrated these consistency mechanisms together in the evolution processes.

Among the different criteria evaluated, Dynamic PRISMA is the only one that enables the specification of architectural models supporting both dynamic reconfiguration and dynamic type evolution, either reactively, proactively or both, enabling the full range of change operations (additions, removals, updates, linkings, unlinkings, and exclusions), introspection features, and integrating mechanisms to guarantee the consistency of the system before and after dynamic changes (state transfer, safe stopping, and transactional changes). The approach integrates all of these features while clearly isolating the concerns related to reconfiguration and evolution from the other concerns, thus improving the reuse and maintenance of the evolution-related concerns.

9.1.3 Evaluation of the research

This section evaluates to which extent the original requirements of the thesis have been satisfied and evaluates how the research has been performed.

On the whole, the main goal of the thesis described in section 1.2 "Overall Aim and Objectives" has been achieved: a framework has been designed that enables the specification and development of architecture-based software systems capable of changing at runtime their structure (i.e. the architecture) and behaviour (i.e. the types). This thesis has described the specification language for describing such systems as well as the supporting mechanisms for developing such systems.

This goal has been achieved by addressing the different objectives stated in section 1.2: (1) the identification of the design strategies and mechanisms enabling dynamic change and their integration in the underlying semantics of the approach; (2) the combination of the two levels of dynamism, by identifying the mechanisms that they have in common; (3) the definition of a set of language constructs and evolution services to describe reconfiguration plans and type evolution requests at a high-abstraction level, which internally use the previously identified mechanisms for dynamic change; (4) the identification and isolation of the evolution concerns into different aspects;

and (5) separation of user-defined behaviour and automatically generated behaviour, to enable its integration in a MDD approach.

Regarding the research methodology used to produce the results, and as explained in section 1.3 (see page 30), a method consistent with the principles of design science has been followed. Next, the results of this work are evaluated through the lens of the design-science guidelines presented by Hevner et al. (Hevner et al., 2004).

The design artefact that has been the subject of study in this work, as stated in the objectives, is Dynamic PRISMA: a framework to support dynamic reconfiguration and type evolution.

- **Problem relevance**. The amount of interest and research related to dynamic change issues from the areas of software architectures, autonomic computing, dynamic product lines, self-managed systems, and self-adaptive systems, testifies the relevance of the work. The areas of application of the work range from the emerging development of autonomous, self-managed systems to the development of mission-critical and/or highly-available systems with flexible requirements.
- **Research rigour**. The work is presented from a technology-independent perspective, at a level of abstraction that allows appreciating the complexity of the processes without technical details. The semantics of the dynamic type evolution processes have been described in terms of typed graph transformations, which provide a great conceptual rigour.
- **Design as a search process**. The different specification languages, techniques, and strategies proposed in the literature have been evaluated, looking for the suitability to the problem addressed. As a result of this study, several key properties have been identified and included in the design of the proposed solution, together with innovative, creative properties (such as asynchronous type evolution, instance-level transformations, or artefact-oriented evolutions).
- **Design as an artefact**. As a result of the research, an artefact has been designed. This artefact is Dynamic PRISMA, a framework supporting the specification of systems that are dynamically reconfigurable and evolvable. It describes the reconfiguration and type evolution processes supporting these systems. It has been instantiated to evaluate its feasibility and get feedback from its realization: it has been implemented in the middleware of PRISMA, PRISMANET. However, since the details about this implementation have not been included in the thesis, the work presented is weak in this point.

- **Design evaluation**. The evaluation of Dynamic PRISMA has been performed through the implementation of a case study, the Agrobot Vision system. This provided a better understanding of the problem and feedback to improve the quality of the framework. However, for realistic purposes, the approach should be evaluated in other systems, such as non-stopping manufacturing systems, cloud computing infrastructures (which are required to be highly available), or highly-used social digital ecosystems (e.g. what would happen if large web ecosystem such as facebook or twitter require to introduce reconfigurations or type evolutions?). These are examples of realistic systems that could be used to evaluate the approach.
- **Research contributions**. As a result of the research work, some innovative concepts and techniques have emerged. The contributions have been discussed in section 9.1.1 and evaluated in section 9.1.2, so they are not described again.
- **Research communication**. The results of this research have been presented and discussed on distinct peer reviewed forums, which have provided valuable feedback to improve the work and disseminate the main contributions (see section 9.2).

9.2 Results of the PhD

Part of the results presented in thesis have been presented and discussed before on distinct peer-reviewed forums, with both international as national impact. Overall, the results of this thesis have been published in two international journals, two book chapters, ten international conferences and workshops, ten national conferences, and two technical reports.

Table 4 summarizes the most relevant publications. This table only contains the publications that have been included in acknowledged rankings, have been cited or have been published by relevant editorials. This table includes the following information:

- **Ref.**: A number that identifies the publication in the list of publications.
- **Conference Name:** the conference where the paper has been accepted,
- **Publisher:** the publisher of the conference proceedings,

- **Ranking:** the ranking of the conference or journal, according to CORE[48] 2009 classification, and
- **Cites:** the number of citations to the paper, according to Google Scholar[49].

Ref.	Conference Name	Publisher	Ranking	Cites
1)	*Special Issue*	*Informatica*	CORE: C	*In press*
3)	SOAR 2010 *Extended papers*	*Springer LNCS*	*(1st edition)*	1
4)	*Book chapter*	*Springer IFIP*	–	5
5)	WICSA/ECSA 2009	IEEE	CORE: A	4
7)	WASELF 2009	*Sistedes*	–	2
8)	ICECCS 2009	IEEE	CORE: A	5
9)	CSMR 2009	IEEE	CORE: C	2
10)	ECSA 2008	*Springer LNCS*	CORE: A	2
11)	ECSA 2007	*Springer LNCS*	CORE: A	6
12)	CBSE 2007	*Springer LNCS*	CORE: A	6
14)	NET Tech 2005	*Union Agency*	CORE: C	16
23)	JISBD 2005	*Thompson*	–	5

Table 4. Most relevant publications

Next, the full list of publications is listed below[50].

International Journals

1) **C. Costa-Soria**, J. Pérez, J.A. Carsí. *An Aspect-Oriented Approach for Supporting Autonomic Reconfiguration of Software Architectures*. Special Issue on Autonomic and Self-Adaptive Systems, Informatica (Slovenia), vol. 35, issue 1, pp. 15-27. February 2011. ISSN 0350-5596.

[48] CORE (COmputing Research Education): http://core.edu.au
[49] http://scholar.google.com (visited 10th February 2011).
[50] Some publications are in Spanish. In these cases, the reference includes the title in both Spanish, to locate the publication, and in English, to facilitate its understanding.

2) N. Ali, J. Pérez, **C. Costa**, I. Ramos, J.A. Carsí. *Replicación distribuida en arquitecturas software orientadas a aspectos utilizando ambientes ("Distributed Replication in Aspect-Oriented Software Architectures Using Ambients")*. IEEE Latin America Transactions, Special Edition JISBD'06, vol. 5, issue 4, pp. 231-237. IEEE Region 9, July 2007. ISSN 1548-0992 *(in Spanish)*

Book Chapters

3) **C. Costa-Soria**, R. Heckel. *Modelling the Asynchronous Dynamic Evolution of Architectural Types*. Weyns, D.; Malek, S.; De Lemos, R.; Andersson, J. (eds.): Self-Organizing Architectures. Lecture Notes on Computer Science Series, vol. 6090, pp. 198-229. Springer-Verlag, Berlin Heidelberg, July 2010.

   (Revised and Extended best papers from SOAR'09 Workshop)

4) N. Ali, J. Pérez, **C. Costa**, I. Ramos, J.A. Carsí. *Mobile Ambients in Aspect-Oriented Software Architectures*. K. Sacha (ed.): Software Engineering Techniques: Design for Quality. IFIP Series, vol. 227, pp. 37-48. Springer, October 2006.

International Conferences & Workshops

5) J. Pérez, J. Díaz, **C. Costa-Soria**, J. Garbajosa. *Plastic Partial Components: A Solution to Support Variability in Architectural Components*. Joint 8th Working IEEE/IFIP Conference on Software Architecture & 3rd European Conference on Software Architecture (WICSA/ECSA 2009), pp. 221-230. Cambridge, UK, 14-17 September. IEEE, 2009

   **Conference Ranking CORE'09: A*

6) **C. Costa-Soria**, R. Heckel. *Formalizing the Asynchronous Evolution of Architecture Patterns*. Workshop on Self-Organizing Architectures (SOAR'09), held at the Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA 2009). Cambridge, UK, 14th September 2009.

7) **C. Costa-Soria**, J. Pérez, J.A. Carsí. *An Aspect-Oriented Approach for Supporting Autonomic Reconfiguration of Software Architectures*. 2nd Workshop on Autonomic and SELF-adaptive Systems (WASELF'09). San Sebastián, Spain, 8th September 2009.

8) **C. Costa-Soria**, D. Hervás-Muñoz, J. Pérez, J.A. Carsí. *A Reflective Approach for Supporting the Dynamic Evolution of Component Types*. 14th

IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'09), pp. 301-310. Potsdam, Germany, 2-4 June 2009.

**Conference Ranking CORE'09: A. Ranking CSCR: 0.88*

9) **C. Costa-Soria**, J. Pérez, J.A. Carsí. *Handling the Dynamic Reconfiguration of Software Architectures using Aspects*. 13th IEEE European Conference on Software Maintenance and Reengineering (CSMR'09), pp. 263-266. Kaiserslautern, Germany, 24-27 March 2009.

   **Conference acceptance ratio: 30%. Ranking CiteSeer: 0.36 (top 59.29%). Ranking CORE'09: C*

10) **C. Costa**, J. Pérez, J.A. Carsí. *Managing Dynamic Evolution of Architectural Types*. Morrison, R., Balasubramaniam, D., Falkner, K. (eds.): 2nd European Conference on Software Architecture (ECSA'08). Lecture Notes on Computer Science, vol. 5292, pp. 281-289. Springer, Heidelberg, 2008.

    **Conference acceptance ratio: 28%. Ranking CORE'09: A*

11) **C. Costa**, N. Ali, J. Pérez, J.A. Carsí, I. Ramos. *Dynamic Reconfiguration of Software Architectures Through Aspects*. Oquendo, F. (ed.): First European Conference on Software Architecture (ECSA'07). Lecture Notes on Computer Science, vol. 4758, pp. 279-283. Springer, Heidelberg, 2007.

    **Conference acceptance ratio: 30%. Ranking CORE'09: A*

12) **C. Costa**, J. Pérez, J.A. Carsí. *Dynamic Adaptation of Aspect-Oriented Components*. H.W. Schmidt, I. Crnkovic, G.T. Heineman, J.A. Stafford (eds.): 10th International Symposium on Component-Based Software Engineering (CBSE'07). Lecture Notes on Computer Science, vol. 4608, pp. 49-65. Springer, Heidelberg, 2007.

    **Conference acceptance ratio: 22%. Ranking CORE'09: A*

13) **C. Costa**, N. Ali, C. Millán, J.A. Carsí. *Transparent Mobility of Distributed Objects using .NET*. 4th International Conference on .NET Technologies. Pilsen, Czech Republic, June 2006.

    **Conference Ranking CORE'09: C (under the acronym C#)*

14) J. Pérez, N. Ali, **C. Costa**, J.A. Carsí, I. Ramos. *Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*. 3rd

International Conference on .NET Technologies. Pilsen, Czech Republic, June 2005.

**Conference Ranking CORE'09: C (under the acronym C#)*

National Conferences & Workshops

15) **C. Costa-Soria**, J. Pérez, J.A. Carsí, D. Alonso, F. Ortiz, J.A. Pastor. *Reconfiguración Dinámica de Arquitecturas Software Aplicada a la Tolerancia a Fallos ("Dynamic Reconfiguration of Software Architectures Applied to Fault-Tolerance").* 3rd International Workshop on Autonomic and Self-Adaptive Systems (WASELF'10). Valencia, Spain, September 2010 *(in Spanish).*

16) **C. Costa**, J. Pérez, J.A. Carsí. *Soporte a la Evolución Dinámica de Tipos Arquitectónicos ("Support for the Dynamic Evolution of Architectural Types").* Workshop on Autonomic and Self-Adaptive Systems (WASELF'08). Gijón, Spain, October 2008 *(in Spanish).*

17) J. Pérez, **C. Costa**, J.A. Carsí, I. Ramos. *Verificación de Modelos Arquitectónicos Orientados a Aspectos ("Verification of Aspect-Oriented Architectural Models").* XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07), pp. 167-176. Zaragoza, 11-14 September 2007 *(in Spanish).*

18) J. Pérez, **C. Costa**, J.A. Carsí, I. Ramos. *PRISMA CASE.* XII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'07). Zaragoza, 11-14 September 2007 *(Demo, in Spanish).*

19) **C. Costa**, J. Pérez, J.A. Carsí. *Hacia la construcción de arquitecturas software dinámicas ("Towards the building of dynamic software architectures").* V Jornadas DYNAMICA, pp. 109-120. Valencia, 23-24 November, 2006 *(in Spanish).*

20) M.E. Cabello, **C. Costa**, I. Ramos. *Arquitectura software orientada a aspectos de un sistema experto multirazonamiento para tareas de diagnóstico ("Aspect-Oriented Software Architecture of a Multi-Reasonament Expert System for Diagnosis Tasks").* XIX Congreso Nacional y V Congreso Internacional de Informática y Computación. Chiapas, México, October 2006 *(in Spanish).*

21) **C. Costa**, J. Pérez, J.A. Carsí. *Hacia la reconfiguración dinámica de arquitecturas software orientadas a aspectos ("Towards the Dynamic Reconfiguration of Aspect-Oriented Software Architectures").* IV Taller de Desarrollo de Software Orientado a Aspectos (DSOA'06). Servicio de

publicaciones de la Universidad de Extremadura, Informe técnico TR 24/06, pp. 35-40. Sitges, Spain, 3 October 2006 *(in Spanish)*.

22) N. Ali, J. Pérez, **C. Costa**, I. Ramos, J.A. Carsí. Replicación distribuida en arquitecturas software orientadas a aspectos utilizando ambientes *("Distributed Replication in Aspect-Oriented Software Architectures using Ambients")*. XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD'06), pp. 379-388. Sitges, October 2006 (in Spanish).

23) **C. Costa**, J. Pérez, N. Ali, J.A. Carsí, I. Ramos. PRISMANET middleware: Soporte a la Evolución Dinámica de Arquitecturas Software Orientadas a Aspectos *("PRISMANET middleware: Support to the Dynamic Evolution of Aspect-Oriented Software Architectures")*. X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05), pp. 27-34. Granada, September 2005 (in Spanish).

24) N. Ali, J. Pérez, **C. Costa**, J.A. Carsí, I. Ramos. *Implementation of the PRISMA Model in the .NET Platform*. II Jornadas DYNAMICA (DYNamic and Aspect-Oriented Modelling for Integrated Component-based Architectures), held with JISBD'04. Malaga, 11 November 2004.

Ms. Science Thesis & Technical Reports

25) D. Hervás-Muñoz. *Evolución dinámica de componentes orientados a aspectos en .NET ("Dynamic Evolution of Aspect-Oriented Components in .NET")*. Ms Science Thesis. Supervisors: **C. Costa-Soria**, J.A. Carsí Cubel. Faculty of Computer Science, Universidad Politécnica de Valencia, February 2009 *(in Spanish)*.

26) S. Aliaga-Varea. *Reconfiguración dinámica de arquitecturas software orientadas a aspectos mediante la plataforma .NET ("Dynamic Reconfiguration of Aspect-Oriented Software Architectures using the .NET platform")*. Ms. Science Thesis. Supervisors: **C. Costa-Soria**, J.A. Carsí Cubel. Faculty of Computer Science, Universidad Politécnica de Valencia, September 2008 *(in Spanish)*.

27) **C. Costa-Soria**. *Estudio e implementación de un modelo de arquitecturas orientado a aspectos y basado en componentes sobre tecnología .NET*. ("*Study and Implementation of an Aspect-Oriented Component-Based Architecture Model on .NET Technology")*. Technical report DSIC-II/11/05. Dept. of Information Systems and Computation, Universidad Politécnica de Valencia, April 2005 *(in Spanish)*.

9.3 Further research

As any other research, the content of this thesis is under continuous development. This section is dedicated to present further work and new research areas that may allow other researchers to eventually produce more useful knowledge and progress in the area.

9.3.1 Model-Driven Development support for evolvable systems

Although there is a broad range of approaches covering different facets of dynamic reconfiguration and type evolution (e.g. specification, formal analysis, consistency management, runtime support, ...), very few works have covered the support for Model-Driven Development (MDD) of evolvable systems. MDD support is interesting because it would facilitate the specification, modelling and code-generation of dynamically evolvable systems.

The future support for MDD has been taken into account during the development of the concepts presented in this thesis. This thesis has not only addressed the definition of a framework to support dynamic reconfiguration and type evolution, but also the identification of which parts of the evolution infrastructure can be automatically generated to free the software architect from the burden of dealing with low-level details. The next step is to integrate this framework, *Dynamic PRISMA*, in the MDD process of PRISMA (see section 2.4.5, page 55).

This involves:

- Extend the PRISMACase tool to integrate the concepts introduced by *Dynamic PRISMA* (i.e. *reconfigurable* composite types, *Reconfiguration Analysis* aspects, *evolvable* architectural types, and manipulation of both *SimpleSpec* and *CompositeSpec* type reifications), supporting both the graphical and textual specification of these concepts.
- Modify the existing code generation patterns to include the *Evolver* component (i.e. for supporting the autonomic reconfiguration of composite types) and *Type Meta-instances* (i.e. for supporting asynchronous reflective evolution of either simple or composite types) in those architectural types that have been defined as *evolvable*. Currently, this is done manually, by modifying the code that is generated from PRISMA models.
- Develop a visual interface to manage running evolvable architecture-based systems, which will interact with the evolution and reconfiguration ports provided by evolvable architectural types,

allowing their introspection and modification at runtime. Currently this is done by means of the ADL language.

Further research work is required to define graphical models that describe proactive evolution and reconfiguration specifications, and to analyse the dynamic properties of evolvable systems.

9.3.2 Proactive non-programmed evolutions

An area of research that still requires further work and study is the support for proactive, non-programmed/generative evolutions. This thesis has been focused on the support for dynamic changes, both in a reactive (i.e. ad-hoc) or in a proactive (i.e. programmed) way. Reactive dynamic changes are introduced by means of the direct invocation of evolution/reconfiguration services. Proactive dynamic changes are defined by means of simple Event-Condition-Action (ECA) policies. These ECA policies are specified at design time, or introduced manually at runtime by means of dynamic type evolution.

However, to truly build self-managing systems, proactive dynamic changes should not be explicitly defined by the architect, but automatically produced according to high-level goals. That is, the long-term objective is to provide self-managed systems with support for *proactive non-programmed evolutions* (see 3.3.4.2, page 72). The challenge is then to define: (i) *what* are high-level goals, and (ii) *how* to transform these high-level (possibly abstract) goals to specific reconfiguration or evolution actions. Emerging works are exploring the automatic task synthesis from high-level goals (Sykes et al., 2008), or the use of digital evolution techniques (McKinley et al., 2008). However, current efforts are still immature and so computational intensive, which cannot be used at runtime.

9.3.3 Definition of evolution constraints

An architectural type (or by extension, the entire software system) can be entirely redefined at runtime by means of dynamic type evolution. However, in certain cases, it may be interesting to limit which parts of an architectural type (or a set of architectural types) can be evolved or not. For instance, this could be interesting to preserve some architectural design decisions and to avoid the removal of critical functional elements. Thus, two kinds of constraints would coexist in an ADL specification: these constraints that can be changed at runtime as any other part of the specification, and those that cannot be changed and that would limit the evolution functionality. This would require the adaptation of the evolution and reconfiguration planning mechanisms to be constraint-aware.

9.3.4 Coordination of decentralized Evolvers

Another interesting area of research is the exploration of self-organization strategies to enable different Evolvers (i.e. the evolution managers from different composite instances) to coordinate their evolutions. This is needed to allow the Evolvers to solve together the evolution conflicts that may result from their independent evolutions. For instance, a composite instance may be interested in removing a functionality that it provides, whereas other adjacent instances could need it.

Dynamic PRISMA implements a decentralized evolution model, motivated by the goal of increasing the scalability and autonomy of evolvable systems. The premise of this approach is that a software system is not evolvable by itself: it is evolvable as a result of the architectural types it is made of. Thus, the evolution of a software system is not centrally managed, but distributed among the different architectural types it is composed of, by means of Evolver components. Therefore, a coordination strategy among the different Evolvers is needed that enables the independent evolution of the composites that these Evolvers manage, but while preserving the quality attributes of the whole system. Some of the early findings about this ongoing work can be found in (Costa-Soria et al., 2011).

9.3.5 Formal analysis

The formal basis of Dynamic PRISMA advantages non-formal approaches in that some advanced analyses can be performed to automate some functions, such as the generation of state transfer functions, the evaluation of semantic compatibility, or the verification of system properties. These kinds of analysis have not been included in the thesis and have been left as a future work. Next they are briefly introduced.

Automatic generation of state transfer functions

When an instance is going to be updated by a new version, or replaced by a different version, its state must be migrated from the old version to the new version. This is carried out by means of state transfer functions (see page 80 for more details). These functions are manually defined by the developer or the architect.

Since each architectural type has its behaviour precisely described (i.e. in terms of the PRISMA AOADL), a potential future work is the automatic generation of the state transformation functions, by means of the realization of behavioural analyses to map the old attributes to the new ones.

Automatic evaluation of semantic compatibility among types

When a type is going to be updated by a new version, replaced by a new one, or introduced in a system, only the syntactic compatibility is evaluated. The evolution agent (e.g. the architect) is responsible of the semantic compatibility of the new type with the existing system (see page 336 for more details).

Another potential future work is the realization of behavioural analyses to evaluate whether two architectural types are semantically compatible and can be connected or not. Although this kind of analysis is feasible (since the behaviour of each architectural type is precisely described, this behaviour can be subject to automated analysis), it is also challenging.

Evaluation of the dependencies and conflicts among reconfiguration transactions

A reconfiguration transaction may be in conflict with another if the former removes an element that the latter introduces again, thus creating a never-ending reconfiguration cycle. A benefit of using formal specifications for describing changes (i.e. reconfigurations transactions) is that these transactions can be analysed and compared looking for potential conflicts or dependencies among them. This can be used to assist the architect in the process of describing Event-Condition-Action rules, or to detect and avoid at runtime the presence of reconfiguration cycles.

Verification of system properties

Another advantage of using formal specifications for describing the structure and behaviour of a system is that model-checking techniques can be used to evaluate whether certain properties are satisfied along the system lifetime or not. Examples of properties that could be evaluated are the following: a component may be instantiated in the architecture (a reachability property), a component will never be removed from the system (a safety property), a connection among two components will be created in any moment of the future (a liveness property), etc.

The challenge relies in the fact that, in dynamically evolvable systems, the set of possible states is very high (in the case of constrained dynamic reconfigurations) or infinite (in the case of dynamic type evolutions). Therefore, it is required the use of model-checking techniques that are efficiently evaluated at runtime whenever a dynamic change is performed, to guarantee that the system properties are preserved after the dynamic change.

9.3.6 Tool support for advanced graph transformations

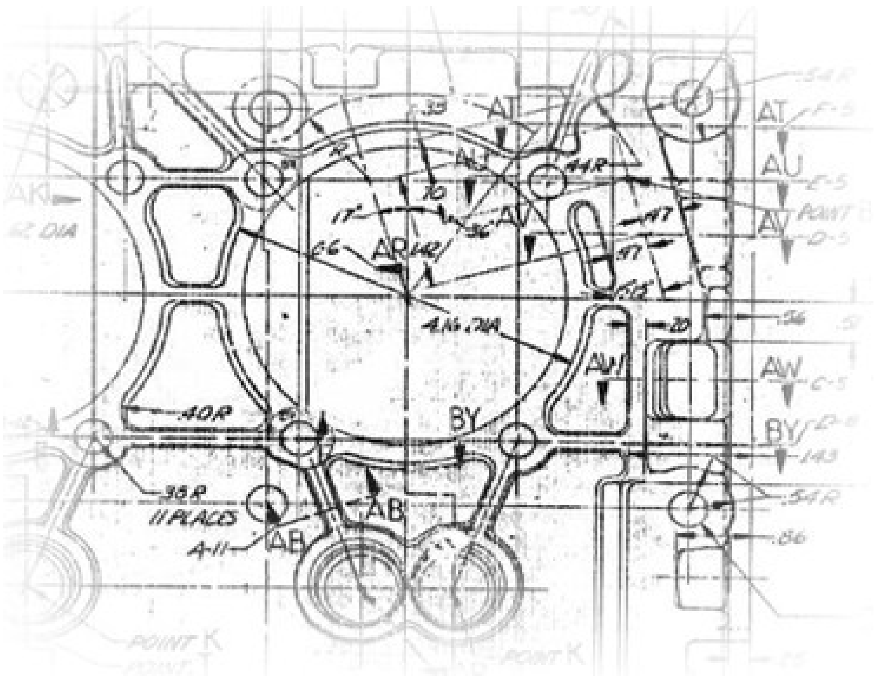
This thesis has explored how the asynchronous evolution semantics can be described by means of typed graph transformations, as a first step towards its formal analysis. However, as a result of this first step, some limitations have been found related to the simulation of graph transformation rules that use advanced concepts, such as complex application conditions or multi-objects[51]. These advanced concepts have been used to describe complex rule preconditions (e.g. that the cardinality of a given architectural type is preserved when creating new instances) or to describe operations on sets of nodes (e.g. quiescing all the instances of a given type, or preserving the existing interactions of a given instance).

The limitations are concerned with the fact that these advanced concepts have no support (or are very difficult to model) in the AGG tool (AGG, 2010), the tool that has been selected to simulate the graph transformations. These limitations are the following: (i) complex application conditions with nested *if-then-else* conditions or that operate on sets of nodes, cannot be defined (this must be done in Java through the use of the internal API); (ii) multi-objects cannot be modelled, rules can only operate on a set of nodes which their exact number is known in advance (we cannot define universally quantified operations); (iii) rules that invoke other rules and provides some parameters, cannot be directly modelled (it requires the usage of auxiliary nodes to pass information among rules); and (iv) the definition of composite rules, which would allow the reuse of rules and the management of different levels of abstraction, is not currently supported. These limitations have been overcome by using the Java interface provided by AGG (for complex application conditions) and assuming that certain conditions apply (for multi-objects). However, an interesting area of research that would improve the expressiveness of graph transformation systems would be considering the support for these advanced features.

[51] An introduction to these advanced concepts can be found in (Heckel, 2006)

PART V

APPENDIXES



APPENDIX A

PRISMA SPECIFICATIONS OF THE VISIONSYSTEM

This appendix provides the complete specifications of the *VisionSystem* composite type (i.e. the subsystem of the *Agrobot* robot that is in charge of capturing and processing images from the environment) using the PRISMA ADL, as well as the specifications of the Reconfiguration and Type Evolution elements.

This appendix is organized as follows. First, section A.1 presents the complete PRISMA specifications of the *VisionSystem* composite type as well as the specification of its instances *RightCamera* and *LeftCamera*. Next, section A.2 presents the specifications of the reconfiguration elements that the *VisionSystem* imports: the *VisionSystemEvolver* and the automatically generated elements. Finally, section A.3 presents the specifications of the type evolution elements that are automatically generated to provide the *VisionSystem* with asynchronous reflective evolution capabilities.

A.1 Specification of the VisionSystem type

A.1.1 Interfaces

A.1.1.1 I_VideoServices

```
Interface I_VideoServices
   newCapturedImage(input capturedImage : Image);
End_Interface I_VideoServices;
```

A.1.1.2 I_ImageProcessingServices

```
Interface I_ImageProcessingServices
   newProcessedImage(input processedImage : Image);
End_Interface I_ImageProcessingServices;
```

A.1.1.3 I_WatchdogEvents

```
Interface I_WatchdogEvents
   validOutput();
   faultyOutput(input failingComponentID: string);
End_Interface I_WatchdogEvents;
```

A.1.2 Data Domains

```
Domains
   // Simple data types
   boolean, natural, integer, double, char, string,

   // Complex data types
   date,    // Date and time functions
   list,    // Array data type
   Image    // Encapsulates an image.
            // Attributes: frameID, imageType & stream (byte array)
End_Domains;
```

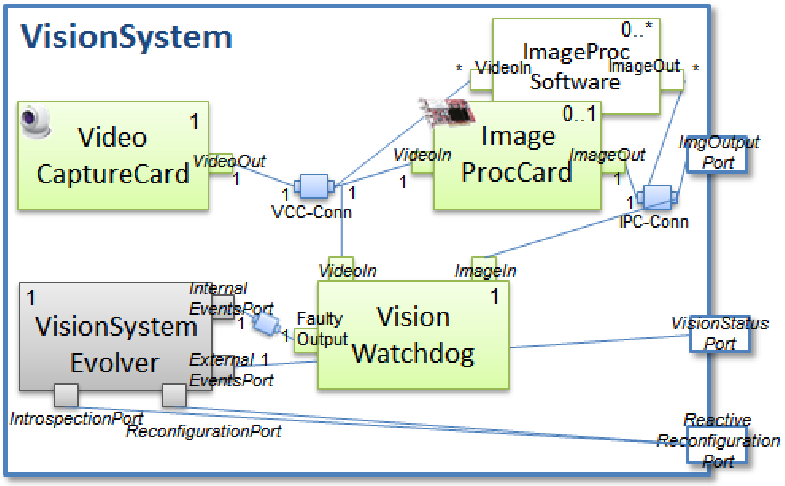
A.1.3 External Functions

```
// Functions provided by external elements
// (COTs, SystemServices, ...)
Image ImageProcessing(Image capturedImage);
   /* Apply an image processing algorithm on the image provided as
   input and returns the transformed image.*/


ByteStream GetImage();
   /* Captures an image from a video capture card */

Boolean TestImageProcessing(Image capturedIMG, Image processedIMG);
   /* Perform a set of simple checkings to verify that processedIMG
      is the a transformation of capturedIMG */

Boolean TestImage(Image imageToTest);
   /* Evaluates if an image is valid */

Integer FCalculateImageProcRatio();
   /* Returns the ratio of images that are processed per second in
the VisionSystem (the higher the best) */

Void Suspend(timeout : integer);
   /* Function provided by the System to suspend a process */
```

A.1.4 Architecture and configurations

A.1.4.1 VisionSystem composite type

PRISMA graphical specification (with Evolver)

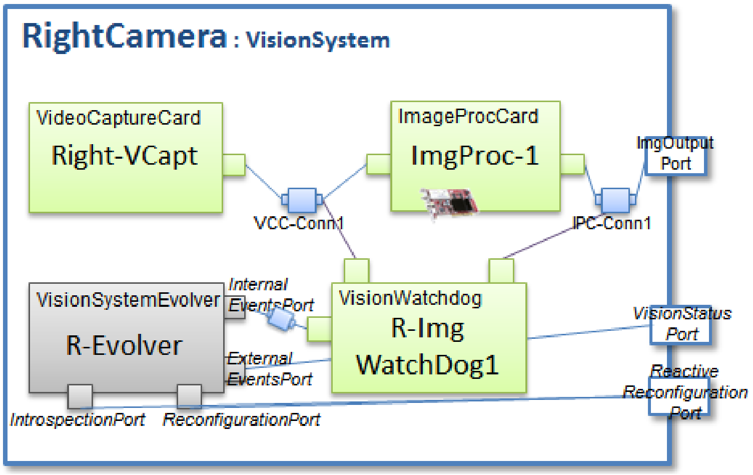


PRISMA ADL textual specification

```
System VisionSystem

   Import Architectural Elements
      VideoCaptureCard(1,1), ImageProcCard(0,1),
      VCC-Conn(1,1), IPC-Conn(1,1),
      ImageProcSoftware(0,*), VisionWatchdog(1,1),
      VisionSystemEvolver(1,1);

   Ports
      ImgOutputPort : I_ImageProcessingServices;
      VisionStatusPort: I_VisionSystemEvents;
      ReactiveReconfigurationPort: I_VisionSystemReconfigServices;
   End_Ports;

   Attachments
      Att_VCC_VCCConn: VideoCaptureCard.VideoOut(1,1) <-->
            VCC-Conn.VideoIn(1,1);
      Att_VCCConn_IPC:
         VCC-Conn.VideoOut(1,1) <--> ImageProcCard.VideoIn(1,1);
      Att_IPC_IPCConn:
         ImageProcCard.ImageOut(1,1) <--> IPC-Conn.ImageIn(1,1);
      Att_VCCConn_ImgMon:
         VCC-Conn.VideoOut(1,1) <--> VisionWatchdog.VideoOutput(1,1);
      Att_IPCConn_ImgMon:
         IPC-Conn.ImageOut(1,1) <--> VisionWatchdog.ImageOutput(1,1);
      Att_VCCConn_IPCSW:
         VCC-Conn.VideoOut(1,1) <--> ImageProcSoftware.VideoIn(1,*);
      Att_IPCSW_IPCConn:
         ImageProcSoftware.ImageOut(1,*) <--> IPC-Conn.ImageIn(1,1);

      // Example of attachment which enables an internal element to
      // interact with the evolver (e.g.to trigger a reconfiguration)
      Att_ImgMon_Evolver: VisionWatchdog.FaultyOutputPort(1,1) <-->
         VisionSystemEvolver.InternalEventsPort(1,1);
   End_Attachments;

   Bindings
      Bin_IPCConn: ImgOutputPort(1,1) <--> IPC-Conn.ImageOut(1,1);
      Bin_Evolver: VisionStatusPort(1,1) <-->
                     VisionSystemEvolver.ExternalEventsPort(1,1);
      Bin_Evolver2: ReactiveReconfigurationPort(1,1) <-->
                     VisionSystemEvolver.IntrospectionPort(1,1);
      Bin_Evolver3: ReactiveReconfigurationPort(1,1) <-->
                     VisionSystemEvolver.ReconfigurationPort(1,1);
   End_Bindings;

   /* Constructor definition */
   new ( frameRate: integer, cameraPosition: string,
         timeout: integer)
   {
      // Only the constructors of the elements that are initially
      // created. ImageProcSoftware is optionally created at runtime

      new VideoCaptureCard(frameRate);
      new ImageProcCard(cameraPosition);
      new VCC-Conn();
      new IPC-Conn();
      new VisionWatchdog(timeout);
      new VisionSystemEvolver(this.ID);

      new Att_VCC_VCCConn(input VideoCaptureCardID: string,
            input VCC-ConnID: string);
      new Att_VCCConn_IPC(input ImageProcCardID: string,
            input VCC-ConnID: string);
      new Att_IPC_IPCConn(input ImageProcCardID: string,
            input IPC-ConnID: string);
      new Att_VCCConn_ImgMon(input VCC-ConnID: string,
            input VisionWatchdogID: string);
      new Att_IPCConn_ImgMon(input IPC-ConnID: string,
            input VisionWatchdogID: string);
      new Att_ImgMon_Evolver(input VisionWatchdogID: string,
            input VisionSystemEvolverID: string);

      new Bin_IPCConn(input ImageProcCardID: string);
      new Bin_Evolver(input VisionSystemEvolverID:string);
      new Bin_Evolver2(input VisionSystemEvolverID:string);
      new Bin_Evolver3(input VisionSystemEvolverID:string);

   }

   /* Destructor definition */
   destroy()
   {
      destroy VideoCaptureCard();
      destroy ImageProcCard();
      destroy ImageProcSoftware();
      destroy VCC-Conn();
      destroy IPC-Conn();
      destroy VisionWatchdog();
      destroy VisionSystemEvolver();

      destroy Att_VCC_VCCConn();
      destroy Att_VCCConn_IPC();
      destroy Att_IPC_IPCConn();
      destroy Att_VCCConn_IPCSW();
      destroy Att_IPCSW_IPCConn();
      destroy Att_VCCConn_ImgMon();
      destroy Att_IPCConn_ImgMon();
      destroy Att_ImgMon_Evolver();

      destroy Bin_IPCConn();
      destroy Bin_Evolver();
      destroy Bin_Evolver2();
      destroy Bin_Evolver3();
   }

End_System VisionSystem;
```

A.1.4.2 Configuration: RightCamera

PRISMA ADL graphical specification



PRISMA ADL textual specification

```
Architectural_Model_Configuration RightCamera =
   new VisionSystem {
      Right-VCapt = new VideoCaptureCard(30);
      ImgProc-1 = new ImageProcCard("right");
      VCC-Conn1 = new VCC-Conn();
      IPC-Conn1 = new IPC-Conn();
      R-ImgWatchDog1 = new VisionWatchdog(90);
      R-Evolver = new VisionSystemEvolver(this.ID);

      att1 = new Att_VCC_VCCConn(Right-VCapt, VCC-Conn1);
      att2 = new Att_VCCConn_IPC(ImgProc-1, VCC-Conn1);
      att3 = new Att_IPC_IPCConn(ImgProc-1, IPC-Conn1);

      att4 = new Att_VCCConn_ImgMon(VCC-Conn1, R-ImgWatchDog1);
      att5 = new Att_IPCConn_ImgMon(IPC-Conn1, R-ImgWatchDog1);
      att6 = new Att_ImgMon_Evolver(R-ImgWatchDog1, R-Evolver);
      bin1 = new Bin_IPCConn(ImgProc-1);
      bin2 = new Bin_Evolver(R-Evolver);
      bin3 = new Bin_Evolver2(R-Evolver);
      bin4 = new Bin_Evolver3(R-Evolver);
   }
```

PRISMA XML Specification

```
<Configuration id=”RightCamera” type=”VisionSystem”>
   <Component id=”Right-VCapt” type=”VideoCaptureCard”>
      <InitParameter>30</InitParameter>
   </Component>
   <Component id=”ImgProc-1” type=”ImageProcCard”>
      <InitParameter>Right</InitParameter>
   </Component>
   <Component id=”R-ImgWatchDog1” type=”VisionWatchdog”>
      <InitParameter>90</InitParameter>
   </Component>
   <Component id=”R-Evolver” type=”VisionSystemEvolver”>
      <InitParameter>RightCamera</InitParameter>
   </Component>

   <Connector id=”VCC-Conn1” type=”VCC-Conn” />
   <Connector id=”IPC-Conn1” type=”IPC-Conn” />

   <Attachment name=”att1” type=”Att_VCC_VCCConn”
      source=”Right-VCapt” target=”VCC-Conn1” />
   <Attachment name=”att2” type=”Att_VCCConn_IPC”
      source=”ImgProc-1” target=”VCC-Conn1” />
   <Attachment name=”att3” type=” Att_IPC_IPCConn”
      source=” ImgProc-1” target=” IPC-Conn1” />
   <Attachment name=”att4” type=” Att_VCCConn_ImgMon”
      source=”VCC-Conn1” target=”R-ImgWatchDog1” />
   <Attachment name=”att5” type=”Att_IPCConn_ImgMon”
      source=”IPC-Conn1” target=”R-ImgWatchDog1” />
   <Attachment name=”att6” type=”Att_ImgMon_Evolver”
      source=”R-ImgWatchDog1” target=”R-Evolver” />

   <Binding name=”bin1” type=”Bin_IPCConn” target=”ImgProc-1” />
   <Binding name=”bin2” type=”Bin_Evolver” target=”R-Evolver” />
   <Binding name=”bin3” type=”Bin_Evolver2” target=”R-Evolver” />
   <Binding name=”bin4” type=”Bin_Evolver3” target=”R-Evolver” />

</Configuration>
```

A.1.4.3 Configuration: LeftCamera

```
Architectural_Model_Configuration LeftCamera =
   new VisionSystem {
      Left-VCapt = new VideoCaptureCard(30);
      ImgProc-2 = new ImageProcCard("left");
      VCC-Conn2 = new VCC-Conn();
      IPC-Conn2 = new IPC-Conn();
      L-ImgWatchDog1 = new VisionWatchdog(90);
      L-Evolver = new VisionSystemEvolver(this.ID);

      att1 = new Att_VCC_VCCConn(Left-VCapt, VCC-Conn2);
      att2 = new Att_VCCConn_IPC(ImgProc-2, VCC-Conn2);
      att3 = new Att_IPC_IPCConn(ImgProc-2, IPC-Conn2);

      att4 = new Att_VCCConn_ImgMon(VCC-Conn2, L-ImgWatchDog1);
      att5 = new Att_IPCConn_ImgMon(IPC-Conn2, L-ImgWatchDog1);
      att6 = new Att_ImgMon_Evolver(L-ImgWatchDog1, L-Evolver);

      bin1 = new Bin_IPCConn(ImgProc-2);
      bin2 = new Bin_Evolver(L-Evolver);
      bin3 = new Bin_Evolver2(L-Evolver);
      bin4 = new Bin_Evolver3(L-Evolver);
   }
```

A.1.5 Components

A.1.5.1 VideoCaptureCard

```
Component VideoCaptureCard

   Integration Aspect import VideoCapture;

   Ports
      VideoOut : I_VideoServices,
         Played_Role VideoCapture.SERVEIMAGE;
   End_Ports;

   new(input frameRate: natural) {
      VideoCapture.begin(frameRate);
   }

   destroy() {
      VideoCapture.end();
   }

End_Component GraphicsCard;
```

A.1.5.2 ImageProcCard

```
Component ImageProcCard

   Integration Aspect import ImageProcCardController;
   Presentation Aspect import ImageProcCardGUI;

   Ports
      VideoIn  : I_VideoServices,
         Played_Role ImageProcCardController.VIDEOCARD;
      ImageOut : I_ImageProcessingServices,
         Played_Role ImageProcCardController.IMAGEANALYZER;
   End_Ports;

   Weavings
      ImageProcCardGUI.showImage(image)
         after
      ImageProcCardController.newProcessedImage(image);
   End_Weavings;

   new(cameraPosition: string) {
      ImageProcCardController.begin(cameraPosition);
      ImageProcCardGUI.begin();
   }

   destroy() {
      ImageProcCardGUI.end();
      ImageProcCardController.end();
   }

End_Component ImageProcCard;
```

A.1.5.3 ImageProcSoftware

```
Component ImageProcSoftware

   Functional Aspect import ImageProcSwController;
   Presentation Aspect import ImageProcCardGUI;

   Ports
      VideoIn  : I_VideoServices,
         Played_Role ImageProcSwController.VIDEOCARD;
      ImageOut : I_ImageProcessingServices,
         Played_Role ImageProcSwController.IMAGEANALYZER;
   End_Ports;

   Weavings
      ImageProcCardGUI.showImage(image)
         after
      ImageProcSwController.newProcessedImage(image);
   End_Weavings;

   new(cameraPosition: string) {
      ImageProcSwController.begin(cameraPosition);
      ImageProcCardGUI.begin();
   }

   destroy() {
      ImageProcCardGUI.end();
      ImageProcSwController.end();
   }

End_Component ImageProcSoftware;
```

A.1.5.4 VisionWatchdog

```
Component VisionWatchdog

   Functional Aspect import ImageMonitoring;
   Ports
      VideoOutput : I_VideoServices,
         Played_Role ImageMonitoring.VIDEOCARD;
      ImageOutput : I_ImageProcessingServices,
         Played_Role ImageMonitoring.IMAGEPROCESSOR;
      FaultyOutput : I_WatchdogEvents,
         Played_Role ImageMonitoring.WATCHDOG;
   End_Ports;

   new(input timeout: natural) {
      ImageMonitoring.begin(input monitorTimeout : natural);
   }

   destroy() {
      ImageMonitoring.end();
   }

End_Component VisionWatchdog;
```

A.1.6 Connectors

A.1.6.1 VCC-Conn

```
Connector VCC-Conn

   Coordination Aspect import VideoForwarding;

   Ports
      VideoIn : I_VideoServices,
         Played_Role VideoForwarding.RECEIVE;
      VideoOut: I_VideoServices,
         Played_Role VideoForwarding.FORWARD;

   End_Ports;

   new() { VideoForwarding.begin(); }
   destroy() { VideoForwarding.end(); }

End_Connector VCC-Conn;
```

A.1.6.2 IPC-Conn

```
Connector IPC-Conn

   Coordination Aspect import ImageForwarding;

   Ports
      VideoIn : I_ImageProcessingServices,
         Played_Role ImageForwarding.RECEIVE;
      VideoOut: I_ImageProcessingServices,
         Played_Role ImageForwarding.FORWARD;

   End_Ports;

   new() { ImageForwarding.begin(); }
   destroy() { ImageForwarding.end(); }

End_Connector IPC-Conn;
```

A.1.7 Aspects

A.1.7.1 VideoCapture

```
Integration Aspect VideoCapture using I_VideoServices
// This aspect provides access to the video capture card

   Attributes
      Constant
         captureTimeout : natural, NOT NULL;
      Variable
         imageCaptured : Image;
         nextFrameNumber : natural;

   Services
      begin(input timeout : natural);
         Valuations
            [begin(timeout)]
            captureTimeout:=timeout;

      out newCapturedImage(input capturedImage : Image);
      // Event that sends a new captured image
         Valuations
            [newCapturedImage(capturedImage)]
            capturedImage:=imageCaptured;

      sleep();
      // Suspends the aspect “timeout” milliseconds

      captureImage();
         Valuations
            [captureImage()]
            imageCaptured:= new Image(nextFrameNumber,
               "CapturedImage", GetImage());
            // Creates a new image object. GetImage is an external
            // function that provides a stream of pixels (an image)
            nextFrameNumber:=nextFrameNumber+1;

      end();

   Played_Roles
      SERVEIMAGE for I_VideoServices ::=
            newCapturedImage!(capturedImage);

   Protocol
      VIDEOCAPTURE ::=  begin(timeout):1 --> CAPTURE;
      CAPTURE ::= end() +
         (  captureImage():10 -->
            SERVEIMAGE_newCapturedImage!(capturedImage):10 -->
            sleep():10
         ) --> CAPTURE;

End Integration Aspect VideoCapture;
```

A.1.7.2 ImageProcSwController

```
Functional Aspect ImageProcSwController
   using I_VideoServices, I_ImageProcessingServices
```

```
    Attributes
       Constant
          spatialPosition : string;
       Variable
          CapturedImage : Image;
          LastProcessedImage : Image;

    Services
       // Initialization service
       begin(input cameraPosition: string);
          Valuations
             (cameraPosition==“right”) or (cameraPosition==“left”)
                [begin(cameraPosition)]
             spatialPosition = cameraPosition;

       // New captured image notification
       in newCapturedImage(input capturedImage: Image);
          Valuations
             [newCapturedImage(capturedImage)]
             CapturedImage = capturedImage;

       // Notifies about the output of a new processed image
       out newProcessedImage(input processedImage : Image);
          Valuations
             [newProcessedImage(processedImage)]
             processedImage = LastProcessedImage;

       // Image processing service
       processImage();
          Valuations
             [processImage()]
             LastProcessedImage = ImageProcessing(CapturedImage);
             // ImageProcessing is an external function

       // Finalization service
       end();

    Played_Roles
       VIDEOCARD for I_VideoServices ::=
          newCapturedImage?(capturedImage);
       IMAGEANALYZER for I_ImageProcessingServices ::=
          newProcessedImage!(processedImage);

    Protocol
       IMAGEPROCESSINGCARDCONTROLLER ::=  begin():1 --> CAPTURE;
       CAPTURE ::= end():10  +
          (VIDEOCARD_newCapturedImage?(capturedImage):10 -->
                processImage():10 -->
             IMAGEANALYZER_newProcessedImage!(processedImage)):10
          ) --> CAPTURE;

End Functional Aspect ImageProcSwController;
```

A.1.7.3 ImageProcCardGUI

```
Presentation Aspect ImageProcCardGUI
    // Only reflects the information shown to the user,
    // but not the design of user interface

    Attributes
```

```
      Constant
         windowTitle : string;
         componentID_label : string;
      Variable
         image_to_show : picture;
         image_label : string;

   Services
      // Field initialization
      begin(input componentName : string);
         Valuations
            [begin(componentName)]
            componentID_label = "Component ID: " + comp_name;
            windowTitle = "Image processing controller";
            image_label = "Image processed";

      // Image showing
      in showImage(input image : list);
         Valuations
            [showImage(image)] image_to_show = image;

      end();

   Protocol
      IMAGEPROCESSINGCARDGUI ::= begin(componentName):1 --> NORMAL;
      NORMAL ::=  showImage?(image):5 + end():10;

End Presentation Aspect ImageProcCardGUI;
```

A.1.7.4 ImageMonitoring

```
// This aspect monitorizes the correctness of image processing
Functional Aspect ImageMonitoring using I_VideoServices,
      I_ImageProcessingServices, I_ VisionSystemEvents

   Attributes
      Constant
         timeout : natural, NOT NULL;
      Variable
         capturedIMG : Image;
         processedIMG : Image;
         processedIMGDelay : natural;
/* This variable counts the number of times non desired data is
received: (i) if only captured images are received, this means that
the image processing component does not work, (ii) if the processed
image does not belong to the captured imaged, this means that the
image processing component does not work correctly */

   Services
      begin(input monitorTimeout : natural);
         Valuations
            [begin(monitorTimeout) ]
            timeout := monitorTimeout;

      in newCapturedImage(input capturedImage : Image);
      // A new image has been captured by the videocard
         Valuations
            [newCapturedImage(capturedImage)]
            capturedIMG:=capturedImage;
      in newProcessedImage(input processedImage : Image);
      // A new image has been processed by the image processing
         Valuations
            [newProcessedImage(processedImage)]
            processedIMG:=processedImage;

      analyzeVideo(output isFaulty : boolean);
      // Checks that capturedImage is a valid image
         Valuations
            [analyzeOutputs(isFaulty)]
            isFaulty:= TestImage(capturedIMG);
            // This function is directly implemented in code

      analyzeOutputs(output isFaulty : boolean);
      // Checks that processedImage is valid and that is equivalent
      // to f2(capturedImage)
         Valuations
            TestImage(processedIMG)==true
            [analyzeOutputs (isFaulty)]
         isFaulty:=TestImageProcessing(capturedIMG,processedIMG);
            // This function is directly implemented in code

      out validOutput();
         // If everything is correct, this event is triggered
      out faultyOutput(input failingComponentID: list);
         // If error, an event is triggered (see protocol section)

      suspend(timeout : integer);
         // This function suspends the current process

      initializeDelayCounter();
         Valuations
            [initializeDelayCounter()]
            processedIMGDelay:=0;

      addDelay();
         Valuations
            [addDelay()]
            processedIMGDelay:=processedIMGDelay+1;

      end();

   Played_Roles
     VIDEOCARD for I_VideoServices ::=
         newCapturedImage?(capturedImage);
     IMAGEPROCESSOR for I_ImageProcessingServices ::=
         newProcessedImage?(processedImage);
     WATCHDOG for I_WatchdogEvents::=
         faultyOutput!(failingComponentID);

   Protocol
   IMAGEMONITORING ::=  begin(monitorTimeout):1 --> GET_CAPTURE;

   // First, get a captured image
   GET_CAPTURE ::= end():10 +
         VIDEOCARD_newCapturedImage?(capturedImage)--> TEST_CAPT;

   // Test if the captured image does not have anomalies
   TEST_CAPT ::=  analyzeVideo(isFaulty):10 -->
```

```
        if (isFaulty==true) then VIDEO_FAULTY else GET_PROCIMG;

   VIDEO_FAULTY ::=
         WATCHDOG_faultyOutput!("VideoCard"):10 --> WAIT;
         // We send a faulty event

   // Get the corresponding processed image
   GET_PROCIMG ::=
      IMAGEPROCESSOR_newProcessedImage?(processedImage) -->
         ( if (processedImage.ID==capturedIMG.ID)
           then TEST_PROC
            // Normal behaviour. Both the captured and processed
            // image have the same ID (they are the same)
           else (IMAGEPROCESSOR_newProcessedImage?(processedImage)
                  --> addDelay()
                  --> GET_PROC_IMG )
         )
      +  (  VIDEOCARD_newCapturedImage?(capturedImage):10 -->
            addDelay():10 ) --> GET_PROC_IMG
         // We have not received yet the processed image we want.
         // We wait for it.

      +  if (processedIMGDelay>20) then
            (addDelay():10 --> PROC_FAULTY);
         // Error, we have reached the maximum number of retries
         // and the desired processed image has not been received

      // We test whether the processed image is correct or not.
      TEST_PROC ::= analyzeOutputs(isFaulty):10 -->
         if (isFaulty==true) then PROC_FAULTY
         else WAIT;

      PROC_FAULTY ::=
         WATCHDOG_faultyOutput!("ImageProcCard"):10 --> WAIT;
         // We send a faulty event

      WAIT ::= validOutput!() --> initializeDelayCounter() -->
      (  suspend(timeout) --> GET_CAPTURE
            // Suspends the monitor until the next test.
      +  VIDEOCARD_newCapturedImage?(captImage) --> WAIT
      +  IMAGEPROCESSOR_newProcessedImage?(procImage) --> WAIT
      );
            // In a waiting state, all new images are discarded.

End Functional Aspect ImageMonitoring;
```

A.1.7.5 VideoForwarding

```
Coordination Aspect VideoForwarding using I_VideoServices

   Services
      begin();
      in/out newCapturedImage(input capturedImage : Image);
      end();

   Played_Roles
     RECEIVE for I_VideoServices ::=
         newCapturedImage?(capturedImage);
     SEND for I_VideoServices ::=
         newCapturedImage!(capturedImage);

   Protocol
      IMAGEFORWARDING ::= begin():1 --> IDLE;
      IDLE ::= end():10 +
         RECEIVE.newCapturedImage?(capturedImage):5 --> FORWARD;
      FORWARD ::=
         SEND.newCapturedImage!(capturedImage):5 --> IDLE;

End Coordination Aspect VideoForwarding;
```

A.1.7.6 ImageForwarding

```
Coordination Aspect ImageForwarding using I_ImageProcessingServices

   Services
      begin();
      in/out newProcessedImage(input processedImage : Image);
      end();

   Played_Roles
     RECEIVE for I_ImageProcessingServices ::=
         newProcessedImage?(processedImage);
     SEND for I_ImageProcessingServices ::=
         newProcessedImage!(processedImage);

   Protocol
      IMAGEFORWARDING ::= begin():1 --> IDLE;
      IDLE ::= end():10 +
         RECEIVE.newProcessedImage?(processedImage) --> FORWARD;
      FORWARD ::=
         SEND.newProcessedImage!(processedImage):10 --> IDLE;

End Coordination Aspect ImageForwarding;
```

A.2 Reconfiguration Elements

A.2.1 Interfaces

A.2.1.1 I_VisionSystemEvents

```
Interface I_VisionSystemEvents
   enabledSystem(input instanceID: string);
   disabledSystem(input instanceID: string, input reason: string);
End_Interface I_VisionSystemEvents;
```

A.2.1.2 I_VisionSystemIntrospectionServices

```
Interface I_VisionSystemIntrospectionServices

// ****************************************************************
// Default interface that defines the introspection services
// that will be available for supporting reactive reconfigurations
// on VisionSystem instances.
//
```

```
// Remove the services that are not intended to be available
// outside a System instance.
// ****************************************************************

   typeOf(elementID: string, output typeName: string);
   getConfigurationSpecification(output PRISMAConfigSpec: string);

   getAttachedArchElems(archElemID: string, attachType: string,
      output attachedArchElemIDs: list);
   getConnectionsOfArchElem(archElemID: string,output conns: list);
   getConnectionsByType(connectionType: string,output conns: list);
   isAttachment(connID: string, isAtt: boolean);
   isBinding(connID: string, isBind: boolean);

   getArchElementProperties(instanceID: string,
      output properties: list, output portsList: list);
   getPortProperties(archElemID: string, portName: string,
      output isProvided: boolean, output isRequired: boolean,
      output interface: string, output connectionList: list);
   getArchElementInitializationValues(archElemID: string,
      output initValues: list);
   getAttachmentProperties(connectionID: string,
      output instance1: string, output instance2: string);
   getBindingProperties(connectionID: string,
      output sysPort: string, output archElemID: string);

End_Interface I_VisionSystemIntrospectionServices;
```

A.2.1.3 I_VisionSystemReconfigurationServices

```
Interface I_VisionSystemReconfigurationServices

// ****************************************************************
// Default interface that defines the reconfiguration services
// that will be available for supporting reactive reconfigurations
// on VisionSystem instances.
//
// Remove the services that are not intended to be available
// outside a VisionSystem instance (except those for transaction
// management, which are required to begin/end reconfigurations)
// ****************************************************************

// *** TRANSACTION MANAGEMENT -do not remove- ***
   BeginConfigurationTransaction();
   EndConfigurationTransaction();
   RollbackConfigurationTransaction();

//*** ARCHITECTURAL ELEMENTS ***
   //*** videoCaptureCard ***
   getInstances-videoCaptureCard(output instances: list);
   replace-videoCaptureCard(oldInstanceID: string,
            framerate: natural, output newInstanceID: string);

   //*** imageProcCard ***
   getInstances-imageProcCard(output instances: list);
   create-imageProcCard(cameraPosition: string,
            output newInstanceID:string);
   destroy-imageProcCard(instanceID: string);
   replace-imageProcCard(oldInstanceID: string,
            cameraPosition: string, output newInstanceID: string);
   //*** imageProcSoftware ***
   getInstances-imageProcSoftware(output instances: list);
   create-imageProcSoftware(cameraPosition: string,
            output newInstanceID: string);
   destroy-imageProcSoftware(instanceID: string);
   replace-imageProcSoftware(oldInstanceID: string,
            cameraPosition: string, output newInstanceID: string);

   //*** visionWatchdog ***
   getInstances-visionWatchdog(output instances: list);
   replace-visionWatchdog(oldInstanceID: string, timeout: natural,
            output newInstanceID: string);

   //*** VCC-Conn ***
   getInstances-VCC-Conn(output instances: list);
   replace-VCC-Conn(oldInstanceID: string, output newID: string);

   //*** IPC-Conn ***
   getInstances-IPC-Conn(output instances: list);
   replace-IPC-Conn(oldInstanceID: string, output newID: string);

//*** CONNECTIONS: ATTACHMENTS & BINDINGS ***
   //*** Att_VCCConn_IPC ***
   attach-Att_VCCConn_IPC(VCC-ConnID: string,
         ImageProcCardID: string);
   detach-Att_VCCConn_IPC(VCC-ConnID: string,
         ImageProcCardID: string);

   //*** Att_VCCConn_IPCSW ***
   attach-Att_VCCConn_IPCSW(VCC-ConnID: string,
         ImageProcSoftwareID: string);
   detach-Att_VCCConn_IPCSW(VCC-ConnID: string,
         ImageProcSoftwareID: string);

   //*** Att_IPC_IPCConn ***
   attach-Att_IPC_IPCConn(ImageProcCardID: string,
         IPC-ConnID: string);
   detach-Att_IPC_IPCConn(ImageProcCardID: string,
         IPC-ConnID: string);

   //*** Att_VCCConn_IPC ***
   attach-Att_IPCSW_IPCConn(ImageProcSoftwareID: string,
         IPC-ConnID: string);
   detach-Att_IPCSW_IPCConn(ImageProcSoftwareID: string,
         IPC-ConnID: string);

End_Interface I_VisionSystemReconfigurationServices;
```

A.2.2 Evolver Component

A.2.2.1 User-defined part: VisionSystemEvolver

```
Evolver-Component VisionSystemEvolver
   is partially defined by VisionSystemEvolverMechanisms

   // Reconfiguration policies
   ReconfigurationAnalysis Aspect
      import VisionSystemReconfigurationAnalysis;
```

```
    // Additional user-defined aspects can be imported if needed

    Ports
       // Ports for providing Reactive Reconfiguration Support
       // (Remove if reactive reconfig. is not needed)
       ReconfigurationPort : I_VisionSystemReconfigurationServices,
          Played_Role VisionSystemReconfigurationServices.RECONFPLANS;
       IntrospectionPort : I_SystemInstanceIntrospectionServices,
          Played_Role Monitoring.INTROSPECT;

       // User-defined ports
       InternalEventsPort : I_WatchdogEvents, Played_Role
             VisionSystemReconfigurationAnalysis.INTERNAL-EVENTS;
       ExternalEventsPort : I_VisionSystemEvents, Played_Role
             VisionSystemReconfigurationAnalysis.EXTERNAL-EVENTS;
    End_Ports;

    Weavings
       // User-defined weavings among user-defined aspects here
    End_Weavings;

    // Initialization and Destruction services
    new(cameraPosition: string) {
       VisionSystemReconfigurationAnalysis.begin(cameraPosition);
    }

    destroy() {
       VisionSystemReconfigurationAnalysis.end();
    }

 End_Evolver-Component VisionSystemEvolver;
```

A.2.2.2 Generated part: VisionSystemEvolverMechanisms

```
Component VisionSystemEvolverMechanisms
   is partial

// ***********************************************************
// Automatically-generated partial Evolver specification for
// VisionSystem instances.
//
// This part contains the weavings among the different reconfiguration
// mechanisms. It is regenerated each time the VisionSystem type changes.
// CANNOT BE CHANGED BY THE USER
// ***********************************************************

   ReconfigurationCoordination Aspect
         import VisionSystemReconfigurationServices;
   Monitoring Aspect import MonitoringAspect;
   ReconfigurationEffector Aspect import EffectorAspect;

   Ports
      SystemTypeDescrPort : I_CompositeTypeDescription,
         Played_Role  VisionSystemReconfigurationServices.
                      META-TYPEDESCRIPTION;
   End_Ports;

   Weavings

   //*********************************************************
```

```
//*** Weavings: RECONFIGURATION_ANALYSIS --> MONITORING ***
//*********************************************************
Monitoring.beforeServiceRequest(*, eventName, eventParams)
   insteadOf
VisionSystemReconfigurationAnalysis.beforeEvent(eventName,eventParams);

Monitoring.insteadOfServiceRequest(*, eventName, condition,
         replacingService, eventParams)
   insteadOf
VisionSystemReconfigurationAnalysis.insteadOfEvent(eventName,
         condition, replacingService, eventParams);

Monitoring.afterServiceRequest(*, eventName, eventParams)
   insteadOf
VisionSystemReconfigurationAnalysis.afterEvent(eventName, eventParams);

Monitoring.getAttachedArchElems(archElemID, attachType,
         attachedArchElemIDs)
   insteadOf
VisionSystemReconfigurationAnalysis.getAttachedArchElems(archElemID,
         attachType, attachedArchElemIDs);

Monitoring.typeOf(instance-ID, type)
   insteadOf
VisionSystemReconfigurationAnalysis.typeOf(instance-ID, type);

Monitoring.getArchElementProperties(ID, properties, ports)
   insteadOf
VisionSystemReconfigurationAnalysis.getArchElementProperties(ID,
      properties, ports);

//... More introspection services

Monitoring.getArchElementInstances(“self”, new list[sysID] )
   insteadOf
VisionSystemReconfigurationAnalysis.getSystemInstanceID(sysID);

Monitoring.getArchElementInstances(“VideoCaptureCard”, list-IDs)
   insteadOf
VisionSystemReconfigurationAnalysis.
      getInstances-videoCaptureCard(list-IDs);

//... Same weavings for each architectural type

//*********************************************************
//*** RECONFIGURATION_ANALYSIS --> RECONFIGURATION EFFECTOR ***
//*********************************************************
ReconfigurationEffector.StartElement(instance-ID)
   insteadOf
VisionSystemReconfigurationAnalysis.StartElement(instance-ID);

ReconfigurationEffector.StopElement(instance-ID)
   insteadOf
VisionSystemReconfigurationAnalysis.StopElement(instance-ID);

//*********************************************************
//*** RECONFIGURATION_ANALYSIS --> RECONFIG. COORDINATION *****
//*********************************************************
VisionSystemReconfigurationServices.BeginConfigurationTransaction()
   insteadOf
```

```
VisionSystemReconfigurationAnalysis.TRANSACTION.BEGIN;

VisionSystemReconfigurationServices.EndConfigurationTransaction()
   insteadOf
VisionSystemReconfigurationAnalysis.TRANSACTION.END;

VisionSystemReconfigurationServices.RollBackConfigurationTransaction()
   insteadOf
VisionSystemReconfigurationAnalysis.TRANSACTION.ROLLBACK;

VisionSystemReconfigurationServices.create-ImageProcCard(params,
         newInstanceID)
   insteadOf
VisionSystemReconfigurationAnalysis.
         create-ImageProcCard(params,newInstanceID);

VisionSystemReconfigurationServices.create-ImageProcSoftware(params,
         newID)
   insteadOf
VisionSystemReconfigurationAnalysis.create-ImageProcSoftware(params,
         newID);

VisionSystemReconfigurationServices.destroy-imageProcCard(instanceID)
   insteadOf
VisionSystemReconfigurationAnalysis.destroy-imageProcCard(instanceID);

VisionSystemReconfigurationServices.destroy-imageProcSoftware(instID)
   insteadOf
VisionSystemReconfigurationAnalysis.destroy-imageProcSoftware(instID);

VisionSystemReconfigurationServices.replace-VideoCaptureCard(oldInstID,
         newParams, newInstanceID)
   insteadOf
VisionSystemReconfigurationAnalysis.replace-VideoCaptureCard(oldInstID,
         newParams, newInstanceID);

VisionSystemReconfigurationServices.replace-ImageProcCard(oldInstaID,
         newParams, newInstanceID)
   insteadOf
VisionSystemReconfigurationAnalysis.replace-ImageProcCard(oldInstID,
         newParams, newInstanceID);

VisionSystemReconfigurationServices.
         replace-ImageProcSoftware(oldInstID, newParams, newInstanceID)
   insteadOf
VisionSystemReconfigurationAnalysis.
         replace-ImageProcSoftware(oldInstID,newParams, newInstanceID);

// ... Similar weavings among the VisionSystemReconfigurationAnalysis
// aspect and the VisionSystemReconfigurationServices aspect

//*********************************************************
//*** Weavings: RECONFIGURATION_COORDINATION --> MONITORING ***
//*********************************************************
Monitoring.typeOf(instanceID, typeName)
   insteadOf
ReconfigurationCoordination.typeOf(instanceID, typeName);

Monitoring.getConfigurationSpecification(spec)
   insteadOf
```

```
ReconfigurationCoordination.getConfigurationSpecification(spec);

Monitoring.getArchElementInstances(typeName, instances)
   insteadOf
ReconfigurationCoordination.getArchElementInstances(typeName,
         instances);

Monitoring.getAttachedArchElems(archElemID, attachType,
         attachedArchElemIDs)
   insteadOf
ReconfigurationCoordination.getAttachedArchElems(archElemID,
         attachType, attachedArchElemIDs);

Monitoring.getConnectionsOfArchElem(archElemID, connectionList)
   insteadOf
ReconfigurationCoordination.getConnectionsOfArchElem(archElemID,
         connectionList);

Monitoring.getConnectionsByType(connectionType, connectionList)
   insteadOf
ReconfigurationCoordination.getConnectionsByType(connectionType,
         connectionList);

// ... Rest of Rec.Coord. services weaved to the
//     Monitoring aspect

//***********************************************************
//*** Weavings: RECONFIGURATION_COORDINATION --> REC.EFFECTOR**
//***********************************************************

ReconfigurationEffector.StartElement(elemID)
   insteadOf
ReconfigurationCoordination.StartElement(elemID);

ReconfigurationEffector.StopElement(elemID)
   insteadOf
ReconfigurationCoordination.StopElement(elemID);

ReconfigurationEffector.CreateInstance(typeName,initParams,instanceID)
   insteadOf
ReconfigurationCoordination.CreateInstance(typeName,initParams,instanceID);

ReconfigurationEffector.DestroyInstance(instanceID)
   insteadOf
ReconfigurationCoordination.DestroyInstance(instanceID);

ReconfigurationEffector.Connect(instance1, port1, instance2, port2,
         connectionID)
   insteadOf
ReconfigurationCoordination.Connect(instance1, port1, instance2, port2,
         connectionID);

ReconfigurationEffector.Disconnect(connectionID)
   insteadOf
ReconfigurationCoordination.Disconnect(connectionID);

ReconfigurationEffector.IsSerializableType(typeName, isSerializable)
   insteadOf
ReconfigurationCoordination.IsSerializableType(typeName, isSerializable);
```

```
   // ... Rest of Rec.Coord. services weaved to the
   //     Reconfiguration Effector aspect

   End_Weavings;

End_Component VisionSystemEvolver-Mechanisms;
```

A.2.3 Reconfiguration Analysis Aspect

A.2.3.1 User-defined part: VSReconfigurationAnalysisAspect

```
ReconfigurationAnalysis Aspect VisionSystemReconfigurationAnalysis
   using I_WatchdogEvents, I_VisionSystemEvents
   is partially defined by VisionSystemAnalysisServices

// *****************************************************************
// User-defined aspect which contains the reconfiguration policies
// for VisionSystem instances.
// *****************************************************************

   Attributes
      Constant
         // User-defined constants here
      Variable
         cameraPos: string;
      Derived
         imageProcPerformance: integer,
            derivation: FCalculateImageProcRatio();

   Services
      // Initialization and destruction services
      begin(cameraPosition : string)
         Valuations
         [begin(cameraPosition] cameraPos = cameraPosition;
      end();

      // Interaction with the VisionWatchdog component
      in faultyOutput(failingComponent: string);
      in validOutput();

      // Notification of critical events to external elements
      out disabledSystem(instanceID: string, reason: string);
      out enabledSystem(systemID: string)

   External Functions
      Suspend(timeout : integer); // Suspends the current process
   End_External Functions

   Played_Roles
      INTERNAL-EVENTS for I_WatchdogEvents ::=
         faultyOutput?(failingComponent) + validOutput?();

      EXTERNAL-EVENTS for I_VisionSystemEvents ::=
         disabledSystem!(instanceID, reason) +
         enabledSystem!(instanceID);

   Triggers
      // Example of activation by means of attribute evaluation
      AddImageProcessor() when
         imageProcPerformance < 10;
      // Example of activation by using played_roles
      RepairImageProcessingUnit() when
         {failingComponent==["ImageProcCard"]}
         INTERNAL-EVENTS_faultyOutput?(failingComponent);

      // Example of activation by means of event capturing
      DisableVisionSystem() when
         {eventParams==["VideoCaptureCard"]}
         beforeEvent!("faultyOutput", out eventParams);

   Transactions
   AddImageProcessor():
      // Config. transaction for creating additional image processors
      ADDIMAGEPROCESSOR ::=
         // Gets IDs of instances required for reconfiguration:
         // the connectors, for attaching the new instance. Only one
         // instance of each connector exists.
         VCCConn-ID=VCC-Conn-list[0] -->
         IPCConn-ID=IPC-Conn-list[0] --> RECONF;
      RECONF::=
         create-ImageProcSoftware!(cameraPos, output newImProcID) -->
         attach-Att_VCCConn_IPCSW!(VCCConn-ID, newImProcID,
            output newAttID) -->
         attach-Att_IPCSW_IPCConn!(newImProcID, IPCConn-ID,
            output newAttID) --> END;

   RepairImageProcessingUnit():
      // Configuration transaction for replacing an imageProcCard
      // component by an imageProcSoftware component
      REPAIRIMAGEPROCESSINGUNIT ::=
         // Get IDs of instances subject to changes
         // Only one instance of ImageProcCard,VCCConn and IPCConn
         // is allowed by the System type, so no iterations are needed
         oldImProcCardID = imageProcCard-list[0] -->
         VCCConnID=VCC-Conn-list[0] -->
         IPCConnID=IPC-Conn-list[0] --> RECONF;
      RECONF ::=
         create-ImageProcSoftware!(cameraPos, output newImProcID) -->
         attach-Att_VCCConn_IPCSW!(VCCConnID, newImProcID,
            output newAttID) -->
         attach-Att_IPCSW_IPCConn!(newImProcID, IPCConnID,
            output newAttID) -->
         detach-Att_VCCConn_IPC!(VCCConnID, oldImProcCardID) -->
         detach-Att_IPC_IPCConn!(oldImProcCardID, IPCConnID) -->
         destroy-ImageProcCard!(oldImProcCardID) --> END;

   DisableVisionSystem():
      // Configuration transaction for disabling the entire system
      // in case an unrecoverable error occurs.
      DISABLEVISIONSYSTEM ::=
         EXTERNAL-EVENTS_disabledSystem!(SystemID,
               "An unrecoverable error has occurred") -->
         StopElement!(videoCaptureCard-list[0]) -->
         StopElement!(visionWatchdog-list[0]) -->
         StopElement!(VCC-Conn-list[0]) -->
         StopElement!(IPC-Conn-list[0]) -->
         {imageProcCard-list.Size()>0}
            // 0..1 instances of ImageProcCard are allowed
            StopElement!(imageProcCard-list[0]) -->
         foreach elem in imageProcSoftware-list do
            // 0..* instances of ImageProcSoftware are allowed
               (StopElement!(elem)) -->
         END;

   IncrementalStart(output success: boolean):
      // Tries to restart all the subsystems after a failure
      INCREMENTALSTART ::=
         <success=false> --> RESTART;
      RESTART ::=
         // Replaces the old instances by new, fresh ones.
         getArchElementInitializationValues!(videoCaptureCard-list[0],
            videoCardInitValues) -->
         replace-VideoCaptureCard!(videoCaptureCard-list[0],
            videoCardInitValues[0], output newID)   -->
         replace-VCC-Conn!(VCC-Conn-list[0], output newID) -->
         getArchElementInitializationValues!(visionWatchdog-list[0],
            watchdogInitValues) -->
         replace-VisionWatchdog!(visionWatchdog-list[0],
            watchdogInitValues[0], output newID) -->
         if (imageProcCard-list.Size()>0) then
            replace-ImageProcCard!(imageProcCard-list[0],
               cameraPos, output newID) -->
         foreach elem in imageProcSoftware-list do (
            replace-ImageProcSoftware!(elem, cameraPos, output newID);
         ) -->
         replace-IPC-Conn!(IPC-Conn-list[0], output newID) -->
         WAITING-TEST;
      WAITING-TEST ::=
         ( INTERNAL-EVENTS_validOutput?() --> <success=true>END )
      +  ( INTERNAL-EVENTS_faultyOutput?(failingID) --> ROLLBACK);

   Protocol
      VISIONRECONFANALYSIS ::=  begin(cameraPosition) --> ANALYSIS;

      ANALYSIS ::= end():10
      +  (  AddImageProcessor():10 ) --> ANALYSIS
      +  (  RepairImageProcessingUnit():10 ) --> ANALYSIS
      +  (  DisableVisionSystem():20 ) --> DISABLEDSTATE;

      DISABLEDSTATE ::= Suspend(1200) -->
         incrementalStart(output success) -->
         if (success==false) then DISABLEDSTATE
         else (EXTERNAL-EVENTS_enabledSystem!(SystemID)--> ANALYSIS);

End ReconfigurationAnalysis Aspect
VisionSystemReconfigurationAnalysis;
```

A.2.3.2 Generated part: VisionSystemAnalysisServices

```
ReconfigurationAnalysis Aspect VisionSystemAnalysisServices
   is partial

// ***************************************************************
// Automatically-generated partial Reconfiguration Analysis aspect
// which defines
// - For the Architect: all the services available to define
//   reconfiguration policies for VisionSystem instances.
// - For the System: the hooks that will be weaved with the
//   reconfiguration mechanisms.
```

```
//
// Do not change, since it is regenerated each time the VisionSystem
// type is changed.
// SHOULD NOT BE CHANGED BY THE USER
// ****************************************************************

   Attributes
      Derived
      // Attributes to query the current configuration (read-only)
         systemID: string, derivation:
            getArchElementInstances("self", output list[0]);
         videoCaptureCard-list : list, derivation:
            getInstances-videoCaptureCard(output
               videoCaptureCard-list);
         imageProcCard-list : list, derivation:
            getInstances-imageProcCard(output imageProcCard-list);
         imageProcSoftware-list: list, derivation:
            getInstances-imageProcSoftware(output
               imageProcSoftware-list);
         visionWatchdog-list: list, derivation:
            getInstances-visionWatchdog(output visionWatchdog-list);
         VCC-Conn-list: list, derivation:
            getInstances-VCC-Conn(output VCC-Conn-list);
         IPC-Conn-list: list, derivation:
            getInstances-IPC-Conn(output IPC-Conn-list);

   Services

   // ************************************************************
   //    Introspection services
   // ************************************************************
      out typeOf(instanceID: string, output typeName: string);

      // Introspection of Architectural Elements
      out getArchElementProperties(instanceID: string,
            output properties: list, output portsList: list);
            // If instanceID="self" the properties of System instance
            // are returned
      out getPortProperties(archElemID: string, portName: string,
            output isProvided: boolean, output isRequired: boolean,
            output interface: string, output connectionList: list);
            // If archElemID="self", the properties of System ports
            // are queried

      // Introspection of connections
      out getAttachedArchElems(archElemID: string, attachType: string,
            output attachedArchElemIDs: list);
            // If attachType=="*" returns all the connections of
            // archElemID
      out getConnectionsOfArchElem(archElemID: string,
            output connectionList: list);
            // If instanceID=="*" returns all of its connections
      out getConnectionsByType(connectionType: string,
            output connectionList: list);
      out isAttachment(connID: string, boolean isAtt);
      out isBinding(connID: string, boolean isBind);
      out getAttachmentProperties(connectionID: string,
            output instance1: string, output instance2: string);
      out getBindingProperties(connectionID: string,
            output sysPort: string, output archElemID: string);
   // ************************************************************
   //    Headers of allowed domain-specific reconfiguration actions
   // ************************************************************
      out create-ImageProcCard(cameraPosition: string,
              output newInstanceID : string);
      out create-ImageProcSoftware(cameraPosition: string,
              output newInstanceID : string);
      out destroy-imageProcCard(instanceID : string);
      out destroy-imageProcSoftware(instanceID : string);

      out replace-VideoCaptureCard(oldInstanceID: string,
              frameRate: natural, output newInstanceID : string);
      out replace-ImageProcCard(oldInstanceID: string,
              cameraPosition: string, output newInstanceID : string);
      out replace-ImageProcSoftware(oldInstanceID: string,
              cameraPosition: string, output newInstanceID : string);
      out replace-VisionWatchdog(oldInstanceID: string,
              timeout: natural, output newInstanceID : string);
      out replace-VCC-Conn(oldInstanceID: string,
              output newInstanceID : string);
      out replace-IPC-Conn(oldInstanceID: string,
              output newInstanceID : string);

      out attach-Att_VCCConn_IPC(VCCConn-ID : string, IPC-ID: string);
      out attach-Att_VCCConn_IPCSW(VCCConn-ID:string,IPCSW-ID:string);
      out attach-Att_IPC_IPCConn(IPC-ID : string, IPCConn-ID: string);
      out attach-Att_IPCSW_IPCConn(IPCSW-ID: string,
              IPCConn-ID: string);

      out detach-Att_VCCConn_IPC(VCCConn-ID: string, IPC-ID: string);
      out detach-Att_IPC_IPCConn(IPC-ID: string, IPCConn-ID: string);
      out detach-Att_VCCConn_IPCSW(VCCConn-ID:string,IPCSW-ID:string);
      out detach-Att_IPCSW_IPCConn(IPC-ID: string,IPCConn-ID: string);

   // ************************************************************
   //    Services for Event Interception
   // ************************************************************
      out beforeEvent(eventName: string, output eventParams: list);
      out afterEvent(eventName:string, output eventParams:list);
      out insteadOfEvent(eventName: string, condition: string,
           replacingService: string, output eventParams:list);

   // ************************************************************
   //    Services for Selective Element Starting/Stopping
   // ************************************************************
      out StartElement(instance-ID: string);
      out StopElement(instance-ID: string);

End ReconfigurationAnalysis Aspect
         VisionSystemAnalysisServices;
```

A.2.4 Reconfiguration Coordination Aspect

A.2.4.1 Base Reconfiguration Coordination Aspect

```
ReconfigurationCoordination Aspect

// ************************************************************
// Predefined behaviour for ReconfigurationCoordination aspects,
// which coordinates runtime, transactional reconfigurations
// of System instances.
// CANNOT BE CHANGED BY THE USER
// **************************************************************

   Attributes
      Constant
         systemID: string; // Reference to the System instance

      Variable
         transactionID: string;
         transState: string = "COMMITTED"; // Default value
         archElementsCreated : list;
         archElementsToDestroy : list;
         connectionsToRemove : list;
         connectionsCreated : list;
      Derived
         ArchElemList : list,
            derivation: getArchElementInstances("*", ArchElemList);
         ConnectionsList : list,
            derivation: getConnectionsOfArchElem("*",ConnectionsList);

   External Functions
      NewTransactionalContext(output transactionID: string);
         // Creates a new transactional context
   CheckConsistence(transactionID: string, output transState: string);
         // Checks if the transactional context is valid or invalid
         // (if an error or exception has occurred)
      FinishTransactionalContext(transactionID: string);
         // Destroys a transactional context and makes changes
         // permanent
      SavePRISMASpec(name: string, specification: string);
         // Saves a PRISMA specification (a type or a configuration)
         // in the filesystem
   End_External Functions

   Services
   // *** Transaction Management ***
      in BeginConfigurationTransaction()
         Valuations
            [BeginConfigurationTransaction()]
            // Initialization of transaction management structures
            transState = "ACTIVE";
            archElementsCreated = new list[];
            archElementsToDestroy = new list[];
            connectionsToRemove = new list[];
            connectionsCreated = new list[];

      in EndConfigurationTransaction()
         Valuations
            [EndConfigurationTransaction()]
            transState="COMMITTED";

      in RollbackConfigurationTransaction()
         Valuations
            [RollbackConfigurationTransaction()]
            transState="ROLLBACKED";

   // *** Generic services for Reconfiguration (private) ***
      CreateArchElem(typeName: string, params: list,
            output newID: string);
      DestroyArchElem(typeName: string, id: string);
      ReplaceArchElem(IDToBeReplaced: string,
         newType: string, initializationValues: list,
         output newID: string);
      CreateAttachment(attachmentType: string, sourceAE-ID: string,
         targetAE-ID: string, output attID: string);
      DestroyAttachment(attachmentType: string,
         instance1-ID: string, instance2-ID: string);
      CreateBinding(bindingType: string, archElemID: string,
         output bindingID: string);
      DestroyBinding(bindingType: string, archElemID: string);

   // *** Required services (used as a weaving hook) ***
      out typeOf(instanceID: string, output typeName: string);
      out getConfigurationSpecification(output PRISMASpec: string);
      out getArchElementInstances(typeName: string, instances: list);
      out getAttachedArchElems(archElemID: string, attachType: string,
            output attachedArchElemIDs: list);
      out getConnectionsOfArchElem(archElemID: string,
            output connectionList: list);
      out getConnectionsByType(connectionType: string,
            output connectionList: list);
      out getArchElementProperties(instanceID: string,
            output properties: list, output portsList: list);
      out getPortProperties(archElemID: string, portName: string,
            output isProvided: boolean, output isRequired: boolean,
            output interface: string, output connectionList: list);
      out getAttachmentProperties(connectionID: string,
            output instance1: string, output instance2: string);
      out getBindingProperties(connectionID: string,
            output sysPort: string, output archElemID: string);
      out isBinding(connID: string, isBind: boolean);
      out getStatus(elementID: string, output status: string);
      out getElementsOfStatus(status: string,
            output elemIDList: list);

      in/out StartElement(elemID: string);
      in/out StopElement(elemID: string);
      out CreateInstance(typeName: string, initParams: list,
            output instanceID: string);
      out DestroyInstance(instanceID: string);
      out Connect(instance1: string, port1: string, instance2: string,
            port2: string, output connectionID: string);
      out Disconnect(connectionID: string);
      out IsSerializableType(typeName: string,
            output isSerializable: boolean);
      out SerializeState(instanceID: string, output state: string);
      out CreateInstanceFromSerializedState(typeName: string,
            serializedState: string, output instanceID: string);
      out CanMigrateStateFromOldVersions(typeName: string,
            output canMigrate: boolean);
      out ConvertStateFromPreviousVersion(typeName: string,
            defaultInitValues: list, oldType: string,oldState: string,
            output transformedState: string);

   Played_Roles
      // Communication with the meta-level
      META-TYPEDESCRIPTION for I_CompositeTypeDescription ::=
         getTypeName!(typeName) + getPorts?(portsList) +
         getArchElementTypes!(archElemList) +
         getAttachmentTypes!(attachmentList) +
         getBindingTypes!(bindingList) +
         ... // any service can be invoked

   Preconditions
      BeginConfigurationTransaction()
         if ( transState="COMMITTED" or transState<>"ROLLBACKED");
         // A new transaction cannot be started if there are
         // unfinished transactions
      EndConfigurationTransaction() if (transState="ACTIVE" );
      RollbackConfigurationTransaction()
         if (transState="ACTIVE" or transSate="ROLLBACK");
         // End and Rollback only when a transaction exists

   // Check the constraints of the architectural type
   // Unused parameters are assigned to void variable, '_' for clarity
   // purposes
      CreateArchElem(typeName, params, newID) if
         (  getArchElementInstances!(typeName, output instances) AND
            META-TYPEDESCRIPTION_getArchTypeProperties!
               (typeName, _, _, _, output AEmaxCard) AND
            instances.Size() < AEmaxCard     );
      DestroyArchElem(typeName, id) if
         (  getArchElementInstances!(typeName, output instances) AND
            instances.Contains(id) AND
            META-TYPEDESCRIPTION_getArchTypeProperties!
               (typeName, _, _, output AEminCard, _) AND
            instances.Size() > AEminCard     );
      ReplaceArchElem(IDToReplace, newType, initValues, newID) if
         (  ArchElemList.Contains(IDToReplace) AND
            META-TYPEDESCRIPTION_getArchElementTypes!(output AETypes)
                  AND
            AETypes.Contains(newType)AND
            typeOf(idToReplace, output oldType) AND
            (oldType=newType OR
              (getArchElementInstances!(newType, output inst1) AND
               META-TYPEDESCRIPTION_getArchTypeProperties!
                  (newType, _, _, _, output maxCard) AND
               inst1.Size() < AEmaxCard AND
               getArchElementInstances!(oldType, output inst2) AND
               META-TYPEDESCRIPTION_getArchTypeProperties!
                  (oldType, _, _, output minCard, _) AND
               inst2.Size() > minCard
              )
            )
         );
      CreateAttachment(attachType, srcAE-ID, trgAE-ID, attID) if
         (  ArchElemList.Contains(sourceAE-ID) AND
            ArchElemList.Contains(targetAE-ID) AND
            META-TYPEDESCRIPTION_getAttachmentTypeProperties!(
               attachType, _, _, _, output srcMaxCard, _, _, _,
               output trgMaxCard) AND
            getAttachedArchElems!(srcAE-ID, attachType,
               output attached2src) AND
            attached2src.Size() < srcMaxCard AND
            getAttachedArchElems!(trgAE-ID, attachType,
               output attached2trg) AND
            attached2trg.Size() < trgMaxCard        );
      DestroyAttachment(attachType, srcAE-ID, trgAE-ID) if
```

```
      (  ArchElemList.Contains(sourceAE-ID) AND
         ArchElemList.Contains(targetAE-ID) AND
         META-TYPEDESCRIPTION_getAttachmentTypeProperties!(
            attachType, _, _, output srcMinCard, _, _, _,
            output trgMinCard, _) AND
         getConnectionsByType!(attachType, output connections) AND
         connections.Size()>0 AND
         getAttachedArchElems!(srcAE-ID, attachType,
            output attached2src) AND
         attached2src.Size() > srcMinCard AND
         getAttachedArchElems!(trgAE-ID, attachType,
            output attached2trg) AND
         attached2trg.Size() > trgMinCard
      );
   CreateBinding(bindingType, archElemID, bindID) if
      (  ArchElemList.Contains(archElemID) AND
         getConnectionsByType!(bindingType, output bindings) AND
         META-TYPEDESCRIPTION_getBindingTypeProperties!
            (bindingType, _, _, _, _, output trgMaxCard) AND
         bindings.Size() < trgMaxCard     );
   DestroyBinding(bindingType, archElemID) if
      (  ArchElemList.Contains(archElemID) AND
         getConnectionsByType!(bindingType, output bindings) AND
         META-TYPEDESCRIPTION_getBindingTypeProperties!
            (bindingType, _, _, _, output trgMinCard, _) AND
         bindings.Size() > trgMinCard
      );

Transactions
BeginConfigurationTransaction():
   BEGINCONFIG::=
      NewTransactionalContext(output transactionID) --> END;

CreateArchElem(typeName, params, output newID):
   CREATE::= CreateInstance!(typeName, params, output newID) -->
               CHECK;
   CHECK::= CheckConsistence(output transState) -->
      if {transState="ROLLBACK"}
      then RollbackConfigurationTransaction()
      else archElementsCreated.add(newID) --> END;

CreateAttachment(attachType, srcAE-ID, trgAE-ID, output attID):
   ATTACH::=
      META-TYPEDESCRIPTION_getAttachmentTypeProperties!
         (attachType, _, output srcAEport, _, _, _,
            output trgAEport, _, _) -->
      Connect!(srcAE-ID, srcAEport, trgAE-ID, trgAEport,
         output attID) --> CHECK;
   CHECK::= CheckConsistence(output transState) -->
      if {transState="ROLLBACK"}
      then RollbackConfigurationTransaction()
      else connectionsCreated.add(attID) --> END;

CreateBinding(bindingType, archElemID, output bindID):
   BIND::=
      META-TYPEDESCRIPTION_getBindingTypeProperties!(bindingType,
             output systemPortName, _, output trgAEport, _, _) -->
      Connect!("self", systemPortName, archElemID, trgAEport,
         output bindID) --> CHECK;
   CHECK::= CheckConsistence(output transState) -->
```

```
      if {transState="ROLLBACK"}
      then RollbackConfigurationTransaction()
      else connectionsCreated.add(bindID) --> END;

DestroyArchElem(typeName, id):
   STOP ::= StopElement!(id) -->
      GetStatus!(id, output status) -->
      if {status="Blocked"} then STOPCONNECTIONS else STOP;
   STOPCONNECTIONS ::=
      GetConnectionsOfArchElem!(id, output connectionList) -->
      for each conn in connectionList do ( StopElement!(conn) )
      --> REMOVECONNECTIONS;
   REMOVECONNECTIONS ::=
      for each conn in connectionList do (
         GetStatus!(conn, output status) -->
         if {status="Blocked"} then connectionsToRemove.add(conn)
         else STOPCONNECTIONS
      ) --> CHECK;
   CHECK::= CheckConsistence(output transState) -->
      if {transState="ROLLBACK"}
      then RollbackConfigurationTransaction()
      else archElementsToDestroy.add(id) --> END;

DestroyAttachment(attachType, srcAE-ID, trgAE-ID):
   GETATTID ::= getConnectionsByType!(attachType, connList) -->
      for each conn in connList do (
         getAttachmentProperties!(conn, output archElem1,
            output archElem2) -->
         if {archElem1=srcAE-ID AND archElem2=trgAE-ID}
         then ( <attachmentID=conn> --> STOP )
         else 0
      );
   STOP ::= StopElement!(attachmentID) -->
      GetStatus!(attachmentID, output status) -->
      if {status="Blocked"} then CHECK else STOP;
   CHECK::= CheckConsistence(output transState) -->
      if {transState="ROLLBACK"}
      then RollbackConfigurationTransaction()
      else connectionsToRemove.add(attachmentID)  --> END;

DestroyBinding(bindingType, archElemID):
   GETBIND-ID ::= getConnectionsByType!(bindingType, connList) -->
      for each conn in connList do (
         getBindingProperties!(conn, _, output AE-ID) -->
         if {AE-ID=archElemID} then ( <bindingID=conn> --> STOP )
         else 0
      );
   STOP ::= StopElement!(bindingID) -->
      GetStatus!(bindingID, output status) -->
      if {status="Blocked"} then CHECK else STOP;
   CHECK::= CheckConsistence(output transState) -->
      if {transState="ROLLBACK"}
      then RollbackConfigurationTransaction()
      else connectionsToRemove.add(bindingID)  --> END;

ReplaceArchElem(idToReplace, newType, initValues, output newID):
   STOPOLDELEM ::= StopElement!(idToReplace) -->
      GetStatus!(idToReplace, output status) -->
      if {status="Blocked"} then STOPCONNECTIONS else STOPOLDELEM;
   STOPCONNECTIONS ::=
      GetConnectionsOfArchElem!(idToReplace, output connectionList)
      --> for each conn in connectionList do
            ( StopElement!(conn) )
      --> MIGRATE;
   MIGRATE ::=
      typeOf!(idToReplace, output oldType) -->
      IsSerializableType!(oldType, output isSerializable) -->
      // Check if old state can be migrated
      if {isSerializable=false}
      then (
         // State of old instance cannot be obtained: it is lost
         CreateInstance!(newType, initValues, output newID)
      )
      else (
         // State of old instance can be obtained:
         SerializeState!(idToReplace, output oldState) -->
         // Check if simple instance replacement is performed
         if {oldType=newType}
         then (
            CreateInstanceFromSerializedState!(oldType, oldState,
               output newID)
         )
         else (
            // We are performing type updating
            // Check if new type can convert old state
            CanMigrateStateFromOldVersions!(newType,
               output canMigrate) -->
            if {canMigrate=true}
            then (
               ConvertStateFromPreviousVersion!(newType,initValues,
                  oldType, oldState, output transformedState) -->
               CreateInstanceFromSerializedState!(newType,
                  transformedState, output newID)
            )
            else (
               // New type cannot accept old structures. Then the
               // old state is lost
               CreateInstance!(newType, initValues, output newID)
            )
         )
      )
      --> MIGRATEOLDCONNS;
   MIGRATEOLDCONNS ::=
      for each conn in connectionList do (
         typeOf!(conn, output connType) -->
         isBinding!(conn, output result) -->
         if {result=true} then (
            // If conn is a binding connection, recreate...
            CreateBinding(connType, newID, output newbindID) -->
            DestroyBinding(conn)
         )
         else (
            // If is an attachment connection, recreate...
            getAttachmentProperties!(conn, output archElem1,
               output archElem2) -->
            if {archElem1=idToReplace}
            then CreateAttachment(connType, newID, archElem2,
                    output attID)
            else CreateAttachment(connType, archElem1, newID,
                    output attID) -->
            DestroyAttachment(conn)
         )
      ) --> CHECK;
   CHECK::= CheckConsistence(output transState) -->
      if {transState="ROLLBACK"}
      then RollbackConfigurationTransaction()
      else (   archElementsToDestroy.add(idToReplace) -->
               archElementsCreated.add(newID) ) -->
      END;

EndConfigurationTransaction():
   CHECK::=
      CheckArchitectureConsistency(output isConsistent) -->
      if { isConsistent=true AND transState="ACTIVE" }
         then COMMIT
         else RollbackConfigurationTransaction();
   COMMIT::=
      foreach connID in connectionsToRemove do
         ( Disconnect!(connID) )          -->
      foreach archElemID in archElementsToDestroy do
         ( DestroyInstance!(archElemID) ) -->
      foreach connID in connectionsCreated do
         ( StartElement!(connID) )        -->
      foreach archElemID in archElementsCreated do
         ( StartElement!(archElemID) )    -->
      getElementsOfStatus("passive", passivatedElements) -->
      foreach stoppedElem in passivatedElements do
         (StartElement!( stoppedElem ) )  -->
      getConfigurationSpecification!(output newConfig) -->
      SavePRISMASpec(systemID, newConfig) --> FINISH;
   FINISH::= FinishTransactionalContext(transactionID) --> END;

RollbackConfigurationTransaction():
   ROLLBACK::=
      foreach connID in connectionsCreated do
         ( Disconnect!(connID) )          -->
      foreach archElemID in archElementsCreated do
         ( DestroyInstance!(archElemID) ) -->
      foreach archElemID in archElementsToDestroy do
         ( StartElement!(archElemID) )    -->
      foreach connID in connectionsToRemove do
         ( StartElement!(connID) )        -->
      getElementsOfStatus("passive", passivatedElements) -->
      foreach stoppedElem in passivatedElements do
         (StartElement!( stoppedElem) )   --> FINISH;
   FINISH::= FinishTransactionalContext(transactionID) --> END;

// Checks that all the required ports are bound to provided ports
CheckArchitectureConsistency(output isConsistent: boolean):
   CHECKCONSISTENCY::=
      foreach archElemID in ArchElemList do (
         getArchElementProperties!(archElemID,_, output portsList)
         -->
         foreach port in portsList do (
            getPortProperties!(archElemID, port, output isProvided,
               output isRequired,_, output connectionList) -->
            if { isRequired=true}
               then (
                  if {connectionList.Size()=0}
                  // A disconnected port has been found
```

```
                    then ( <isConsistent:=false> --> END )
                 )
                 else 0
              )
          ) -->
          <isConsistent:=true> --> END;

End ReconfigurationCoordination Aspect;
```

A.2.4.2 Generated part: VisionSystemReconfigurationServices

```
ReconfigurationCoordination Aspect VisionSystemReconfigurationServices
   using I_VisionSystemReconfigurationServices

// ***************************************************************
// Automatically-generated domain-specific aspect for coordinating
// ad-hoc and programmed reconfigurations of VisionSystem instances.
// CANNOT BE CHANGED BY THE USER
// ***************************************************************

   Services
      in begin();
      in end();

   // *** DOMAIN-SPECIFIC RECONFIGURATION SERVICES ***
      //*** Create- Services ***
      in create-ImageProcCard(cameraPosition: string,
            output newInstanceID:string);
      in create-ImageProcSoftware(cameraPosition: string,
            output newInstanceID: string);

      //*** Destroy- Services ***
      in destroy-imageProcCard(instanceID : string);
      in destroy-imageProcSoftware(instanceID : string);

      //*** Replace- Services ***
      in replace-VideoCaptureCard(oldInstanceID: string,
            frameRate: natural, output newInstanceID : string);
      in replace-ImageProcCard(oldInstanceID: string,
            cameraPosition: string, output newInstanceID : string);
      in replace-ImageProcSoftware(oldInstanceID: string,
            cameraPosition: string, output newInstanceID : string);
      in replace-VisionWatchdog(oldInstanceID: string,
            timeout: natural, output newInstanceID : string);
      in replace-VCC-Conn(oldInstanceID: string,
            output newInstanceID : string);
      in replace-IPC-Conn(oldInstanceID: string,
            output newInstanceID : string);

      //*** Attach- Services ***
      in attach-Att_VCCConn_IPC(VCCConn-ID : string, IPC-ID: string);
      in attach-Att_VCCConn_IPCSW(VCCConn-ID:string,IPCSW-ID:string);
      in attach-Att_IPC_IPCConn(IPC-ID : string, IPCConn-ID: string);
      in attach-Att_IPCSW_IPCConn(IPCSW-ID:string, IPCConn-ID:string);

      //*** Detach- Services ***
      in detach-Att_VCCConn_IPC(VCCConn-ID: string, IPC-ID: string);
      in detach-Att_IPC_IPCConn(IPC-ID: string, IPCConn-ID: string);
      in detach-Att_VCCConn_IPCSW(VCCConn-ID:string,IPCSW-ID:string);
      in detach-Att_IPCSW_IPCConn(IPC-ID: string,IPCConn-ID: string);
      //*** Bind- Services ***
      // NONE
      //*** Unbind- Services ***
      // NONE

      // *** Services to query the current configuration (read-only)
      in getInstances-videoCaptureCard(output instances: list);
      in getInstances-imageProcCard(output instances: list);
      in getInstances-imageProcSoftware(output instances: list);
      in getInstances-visionWatchdog(output instances: list);
      in getInstances-VCC-Conn(output instances: list);
      in getInstances-IPC-Conn(output instances: list);

   Played_Roles
      RECONFPLANS for I_VisionSystemReconfiguration ::=
         BeginConfigurationTransaction?() -->
         (  create-imageProcCard?(newID) +
            destroy-imageProcCard?(instanceID) +
            replace-imageProcCard?(oldID, newID) +
            attach-Att_VCCConn_IPC?(VCC-ConnID, IPC-ID) +
            detach-Att_VCCConn_IPC?(VCC-ConnID, IPC-ID) +
            ...
            ... // Rest of available Reconfiguration Services
            ...
            RollbackConfigurationTransaction?() +
            EndConfigurationTransaction?()
         );

   Protocol
      VISIONSYSTEMRECONFIGURATION ::= begin() --> WAITING;
      WAITING ::=
         RECONFPLANS.BeginConfigurationTransaction?() --> RECONFIG;
      RECONFIG ::=
      (  // *** Provided Reconfiguration Services ***
            RECONFPLANS.create-imageProcCard?(cameraPosition, newID)
            --> CreateArchElem("ImageProcCard",
                  new list[cameraPosition], newID)
         +  RECONFPLANS.destroy-imageProcCard?(instanceID)
            --> DestroyArchElem("ImageProcCard",instanceID)
         +  RECONFPLANS.replace-imageProcCard?(oldInstanceID,
               cameraPosition, newInstanceID)
            --> ReplaceArchElem(oldInstanceID,"ImageProcCard",
               new list[cameraPosition], newInstanceID)
         +  RECONFPLANS.attach-Att_VCCConn_IPC?(VCC-ConnID, IPC-ID)
            --> CreateAttachment("Att_VCCConn_IPC", VCC-ConnID,
               IPC-ID, newAttID)
         +  RECONFPLANS.detach-Att_VCCConn_IPC?(VCC-ConnID, IPC-ID)
            --> DestroyAttachment("Att_VCCConn_IPC", VCC-ConnID,
               IPC-ID)
         +  RECONFPLANS.create-imageProcSoftware?(cameraPos, newID)
            --> CreateArchElem("ImageProcSoftware",
               new list[cameraPos], newID)
         +  RECONFPLANS.destroy-imageProcSoftware?(instanceID)
            --> DestroyArchElem("ImageProcSoftware",instanceID)
         +  RECONFPLANS.replace-imageProcSoftware?(oldInstanceID,
               cameraPosition, newInstanceID)
            --> ReplaceArchElem(oldInstanceID, "ImageProcSoftware",
                  new list[cameraPosition], newInstanceID)
         +  RECONFPLANS.attach-Att_VCCConn_IPCSW?(VCC-ConnID,IPCSW-ID)
```

```
            --> CreateAttachment(“Att_VCCConn_IPCSW”, VCC-ConnID,
                IPCSW-ID, newAttID)
         +  RECONFPLANS.detach-Att_VCCConn_IPCSW?(VCC-ConnID,IPCSW-ID)
            --> DestroyAttachment(“Att_VCCConn_IPCSW”, VCC-ConnID,
                IPCSW-ID)
            ...
            ... // Rest of available Reconfiguration Services
            ...
      ) --> RECONFIG
      +  (  // *** Introspection services ***
         +  RECONFPLANS.getInstances-videoCaptureCard?(instances)
            --> getArchElementInstances!(“videoCaptureCard”,instances)
         +  RECONFPLANS.getInstances-imageProcCard?(instances)
            --> getArchElementInstances!(“imageProcCard”,instances)
         +  RECONFPLANS.getInstances-imageProcSoftware?(instances)
            -->getArchElementInstances!(“imageProcSoftware”,instances)
         +  RECONFPLANS.getInstances-visionWatchdog?(instances)
            --> getArchElementInstances!(“visionWatchdog”,instances)
         +  RECONFPLANS.getInstances-VCC-Conn?(instances)
            -->getArchElementInstances!(“VCC-Conn”,instances)
         +  RECONFPLANS.getInstances-IPC-Conn?(instances)
            --> getArchElementInstances!(“IPC-Conn”,instances)
      ) --> RECONFIG
      +  RECONFPLANS.RollbackConfigurationTransaction?()--> WAITING
      +  RECONFPLANS.EndConfigurationTransaction?()--> WAITING;

End ReconfigurationCoordination Aspect
VisionSystemReconfigurationServices;
```

A.2.5 Architecture Monitoring Aspect

```
ArchitectureMonitoring Aspect

// ***************************************************************
// Platform-dependent aspect for monitoring and introspecting
// System instances at runtime.
//
// Here are described the public services that are provided.
// Internal behaviour is provided by low-level implementations.
// CANNOT BE CHANGED BY THE USER
// ***************************************************************

   Attributes
      // All of them private, only shown for descriptive use
      Constant
         systemID: string; // Reference to the System instance
      Variable
         monitoredServices: list;
         architecturalElements: list;
         systemPorts: list;
         attachments: list;
         bindings: list;

   Services
      // **** Introspection services ****
      in typeOf(instanceID: string, output typeName: string);
      in getConfigurationSpecification(output PRISMAConfSpec: string);

      in getArchElementInstances(typeName: string,
            output instances: list);
            // If typeName==”*” returns all the instances
      in getAttachedArchElems(archElemID: string, attachType: string,
            output attachedArchElemIDs: list);
            // If attachType==”*” returns all the connections of
            // archElemID
      in getConnectionsOfArchElem(archElemID: string,
            output connectionList: list);
            // If archElemID==”*” returns all the created connections
      in getConnectionsByType(connectionType: string,
            output connectionList: list);
      in isAttachment(connID: string, isAtt: boolean);
      in isBinding(connID: string, isBind: boolean);

      in getArchElementProperties(archElemID: string,
            output properties: list, output portsList: list);
            // If instanceID=”self” the properties of System instance
            // are returned
      in getPortProperties(archElemID: string, portName: string,
            output isProvided: boolean, output isRequired: boolean,
            output interface: string, output connectionList: list);
            // If archElemID=”self”, the properties of System ports
            // are queried
      in getArchElementInitializationValues(archElemID: string,
            output initValues: list);
      in getAttachmentProperties(connectionID: string,
            output instance1: string, output instance2: string);
      in getBindingProperties(connectionID: string,
            output sysPort: string, output archElemID: string);

      // **** Runtime status ****
      in getStatus(elementID: string, output status: string);
      in getElementsOfStatus(status: string, output elemIDList: list);
         // Return the elements that are in a specific “status”

      // **** Event interception services ****
      in beforeServiceRequest(elemID: string, serviceName: string,
            output params: list);
      in afterServiceRequest(elemID: string, serviceName: string,
            output params: list);
      in insteadOfServiceRequest(elemID: string, serviceName: string,
            condition: string, replacingService: string,
            output params: list);
      in monitoredServices(output serviceList: list);

   Played_Roles
      // Services provided for Ad-hoc reconfiguration
      INTROSPECT for I_SystemInstanceIntrospectionServices ::=
         typeOf?(elementID, typeName) +
         getConfigurationSpecification?(PRISMAConfigSpec) +
         getArchElementInstances?(typeName, instances) +
         getAttachedArchElems?(archElemID, attachType, attachedAEs) +
         ...
         // any service of I_SystemInstanceIntrospectionServices

End ArchitectureMonitoring Aspect;
```

A.2.6 Architecture Effector Aspect

```
ArchitectureEffector Aspect
```

```
// ***********************************************************
// Platform-dependent aspect for changing System instances at runtime.
//
// Here are described the public services that are provided.
// Internal behaviour is provided by low-level implementations.
// CANNOT BE CHANGED BY THE USER
// ***********************************************************

   Services
      // Services for Safe Stopping
      in StartElement(elemID: string);
         // Reach an Active status.
      in StopElement(elemID: string);
         // Reach a Quiescent status. This may require the passivation
         // of neighbours
      in PassivateElement(elemToPassivate: string,
           blockedElement: string);
         // Passivates an element with respect the interactions with
         // another element

      // Basic services for Reconfiguration
      in CreateInstance(typeName: string, initParams: list,
           output instanceID: string);
      in DestroyInstance(instanceID: string);
      in Connect(instance1: string, port1: string, instance2: string,
           port2: string, output connectionID: string);
      in Disconnect(connectionID: string);

      // Auxiliary services for Mobility, Recovery and Updating
      in IsSerializableType(typeName: string,
           output isSerializable: boolean);
      in SerializeState(instanceID: string, output state: string);
      in CreateInstanceFromSerializedState(typeName: string,
           serializedState: string, output instanceID: string);
      in CanMigrateStateFromOldVersions(typeName: string,
           output canMigrate: boolean);
      in ConvertStateFromPreviousVersion(typeName: string,
           oldType: string, oldState: string,
           newRequiredValues: list, output transformedState: string);

End ArchitectureEffector Aspect;
```

A.3 Type Evolution Elements

A.3.1 Data structures

A.3.1.1 Related to simple types

SimpleSpec

```
Data Structure SimpleSpec

   Attributes
      Variable
         typeName: string;
         namespace: string;
         kind: { Component | Connector };
         version: string;
         aspectList: list(AspectInfo);
         weavingList: list(WeavingInfo);
         portList: list(PortInfo);
         constructorList: list(ConstructorInfo);
         destructor: list(DestructorInfo);

         evolutionSteps: list(STEvolutionStep);

   Services
      AddAspect(aspectType: string, parameters: string);
      RemoveAspect(aspectType: string);
      ReplaceAspect(aspectToReplace: string, newAspectType: string,
         newAspectParameters: string);
      AddPort(portName: string, interface: string,
         playedRole: string);
      RemovePort(portName: string);
      AddWeaving(sourceAspect: string, sourceMethod: string,
         sourceParameters: list, weavingType: string,
         targetAspect: string, targetMethod: string,
         targetParameters: list, transfFunctions: list);
      RemoveWeaving(sourceAspect: string, sourceMethod: string,
         weavingType: string, targetMethod: string);
      StopAspect(aspectType: string);
      DefineConstructor(constructorParameters: list(ParamInfo),
         expressions: list(Expression));
      DefineDestructor(expressions: list(Expression));

End_Data_Structure SimpleSpec
```

AspectInfo

```
Data Structure AspectInfo
   Attributes
      aspectName: string;
      typeDefinition: string;
         // Reference to the aspect definition or implementation
      concern: string; // eg. Functional, Presentation, etc.
      interfaces: list(string);
      playedRoles: list(string);
End_Data_Structure AspectInfo;
```

PortInfo

```
Data Structure PortInfo
   Attributes
      portName: string;
      portInterface: string;
      portPlayedRole: string;
      isProvided: boolean;
      isRequired: boolean;
End_Data_Structure PortInfo;
```

WeavingInfo

```
Data Structure WeavingInfo
   Attributes
      sourceAspect: string;   // Aspect to be intercepted
      sourceMethod: string;   // Method to be intercepted
      sourceParameters: list(string);
      weavingType: string;    // Before, After, Instead, InsteadIf
      targetAspect: string;
      targetMethod: string;
      targetParameters: list(string);
      transfFunctions: list(string);
End_Data_Structure WeavingInfo;
```

A.3.1.2 Related to composite types

CompositeSpec

```
Data Structure CompositeSpec

   Attributes
      Variable
         typeName: string;
         namespace: string;
         kind: { System };
         version: string;

         architecturalTypeList: list(AETypeInfo);
         attachmentTypeList: list(AttTypeInfo);
         bindingTypeList: list(BindTypeInfo);
         portList: list(PortInfo);
         constructorList: list(ConstructorInfo);
         destructor: list(DestructorInfo);

         evolutionSteps: list(CTEvolutionStep);

   Services
      AddArchitecturalType(AEName: string, typeDefinition: string,
         minCard: integer, maxCard: integer);
         // aeName: Name of the architectural type in the System
         //          (i.e. an alias).
         // typeDefinition: A reference to the concrete type
         //          specification or implementation.

      AddAttachmentType(attName: string, sourceAE: string,
         srcPort: string, srcMinCard: integer, srcMaxCard: integer,
         targetAE: string, trgPort: string, trgMinCard: integer,
         trgMaxCard: integer);
      AddBindingType(bindName: string, sysPort: string,
         targetAE: string, trgMinCard:integer, trgMaxCard:integer);
      AddPort(portName: string, interface: string);
      RemoveArchitecturalType(AEName: string);
      RemoveAttachmentType(attName: string);
      RemoveBindingType(bindName: string);
      RemovePort(sysPortName: string);
      UpdateArchitecturalType(oldType: string, newType: string,
         minCard: integer, maxCard: integer);
      UpdateArchitecturalTypeCard(AEName: string,
         newMinCard: integer, newMaxCard: integer);
      UpdateAttachmentType(attName: string, srcMinCard: integer,
         srcMaxCard: integer, trgMinCard: integer,
         trgMaxCard:integer);
      UpdateBindingType(bindName: string, trgMinCard: integer,
         trgMaxCard: integer);
      DefineConstructor(constructorParameters: list(ParamInfo),
         expressions: list(Expression));
      DefineDestructor(expressions: list(Expression));
```

```
End_Data_Structure CompositeSpec
```

Architectural Element Info

```
Data Structure AETypeInfo
   Attributes
      name: string; // Name of the architectural type in the System
      minCard: integer;
      maxCard: integer;
      typeDefinition: string; // Reference to the implementation
End_Data_Structure AETypeInfo;
```

AttachmentType Info

```
Data Structure AttTypeInfo
   Attributes
      attName : string;
      srcArchElemType: string; srcAEport: string;
      srcMinCard: integer;
      srcMaxCard: integer;
      trgArchElemType: string; trgAEport: string;
      trgMinCard: integer;
      trgMaxCard: integer;
End_Data_Structure AttTypeInfo;
```

BindingType Info

```
Data Structure BindTypeInfo
   Attributes
      name : string;
      systemPortName: string;
      trgArchElemType: string; trgAEport: string;
      trgMinCard: integer;
      trgMaxCard: integer;
End_Data_Structure BindTypeInfo;
```

A.3.1.3 Constructor and destructor

ConstructorInfo

```
Data Structure ConstructorInfo
   Attributes
      parameters: list(ParamInfo);
      expressions: list(Expression);
End_Data_Structure ConstructorInfo;
```

DestructorInfo

```
Data Structure DestructorInfo
   Attributes
      expressions: list(Expression);
End_Data_Structure DestructorInfo;
```

ParamInfo

```
Data Structure ParamInfo
   Attributes
```

```
      parameterName: string;
      parameterType: string;
End_Data_Structure ParamInfo;
```

Expression

```
Data Structure Expression
   Attributes
      varName: string;        // Name of a variable for assignments
      varType: string;        // Type of the variable
      expression: string;     // Expression

/* This simple structure allows us to encapsulate the following
statements:
      Variable declaration → varType varName;
      Statement → expression;
      Variable assignment → varName = expression;
      Declaration and assignment → varType varName = expression;
*/
End_Data_Structure Expression;
```

A.3.1.4 CTEvolutionStep

```
Data structure CTEvolutionStep
   Attributes
      action : {Add | Remove | Replace | Start | Stop};
      type: { ArchitecturalElement | Attachment | Binding | Port };
      data: { ArchitecturalElementInfo | AttachmentInfo |
              BindingInfo | PortInfo | ReplacementInfo };
End_Data_Structure CTEvolutionStep;
```

A.3.1.5 STEvolutionStep

```
Data structure STEvolutionStep
   Attributes
      action : {Add | Remove | Replace | Start | Stop};
      type: { Aspect | Port | Weaving };
      data: { AspectInfo | PortInfo | WeavingInfo |
              ReplacementInfo };
End_Data_Structure STEvolutionStep;
```

A.3.1.6 ReplacementInfo

```
Data structure ReplacementInfo
   Attributes
      oldAspect : AspectInfo;
      newAspect : AspectInfo;
End_Data_Structure ReplacementInfo;
```

A.3.2 Interfaces

A.3.2.1 I_SimpleTypeDescription

```
Interface I_SimpleTypeDescription

// ****************************************************************
// Interface providing type introspection services for
// simple architectural types (i.e. Components and Connectors)
// ***********************************************************

   getTypeSpecification(output xmlSpec: string);
   getPorts(output portsList: list);
   getAspectTypes(output aspectTypesList: list);
   getWeavings(output weavingList: list);

   getAspectTypeProperties(aspectName: string,
      output typeDefinition: string, output concern: string,
      output interfaces: list, output playedRoles: list);
   getPortProperties(portName: string,
      output isProvided: boolean, output isRequired: boolean,
      output interface: string, output playedRole: string);
   getWeavingProperties(id: string,
      output sourceAspect: string, output sourceMethod: string,
      output sourceParameters: list, output weavingType: string,
      output targetAspect: string, output targetMethod: string,
      output targetParameters: list, output transfFunctions: list);
End_Interface I_SimpleTypeDescription;
```

A.3.2.2 I_CompositeTypeDescription

```
Interface I_CompositeTypeDescription

// ***********************************************************
// Interface providing type introspection services for
// System architectural types.
// ***********************************************************

   getTypeSpecification(output xmlSpec: string);
   getSystemPorts(output portsList: list);

   getArchElementTypes(output archElementList: list);
   getAttachmentTypes(output attachmentList: list);
   getBindingTypes(output bindingList: list);

   getArchTypeProperties(aeName: string,
      output typeDefinition: string, output portList: string,
      output minCard: integer, output maxCard: integer);
      // aeName: Name of the architectural type inside the System
      //          (i.e. an alias).
      // typeDefinition: A reference to the concrete type
      //          specification or implementation.
   getPortProperties(archElemType: string, portName: string,
      output isProvided: boolean, output isRequired: boolean,
      output interface: string);
   getAttachmentTypeProperties(attType: string,
      output srcArchElemType: string, output srcAEport: string,
      output srcMinCard: integer, output srcMaxCard: integer,
      output trgArchElemType: string, output trgAEport: string,
      output trgMinCard: integer, output trgMaxCard: integer);
   getBindingTypeProperties(bindType: string,
      output systemPortName: string,
      output trgArchElemType: string, output trgAEport: string,
      output trgMinCard: integer, output trgMaxCard: integer);

   getConnectionsofType(archElemType: string,
      output attachments: list, output bindings);
   getAttachedTypes(archElemType: string,
      output attachedArchTypes: list);

End_Interface I_CompositeTypeDescription;
```

A.3.2.3 I_SerializableType

```
Interface I_SerializableType

// ***************************************************************
// Services that must provide a type to support the
// Dynamic Updating (state migration), Recovery or Mobility
// ***************************************************************

   SerializeState(output exportedState: string);
   CreateInstanceFromSerializedState(serializedState: string,
      output instanceID: string);

End_Interface I_SerializableType;
```

A.3.2.4 I_IncrementalUpdating

```
Interface I_IncrementalUpdating

// ***************************************************************
// Services that must provide a type to support the
// migration of instances from previous versions.
// ***************************************************************

   ConvertStateFromPreviousVersion(oldTypeName: string,
      oldState: string, newRequiredValues: list,
      output transformedState: string);
   CreateInstanceFromSerializedState(serializedState: string,
      output instanceID: string);

End_Interface I_IncrementalUpdating;
```

A.3.3 Type Description Aspect

A.3.3.1 Data structures: TypeVersion

```
Data Structure TypeVersion
   Attributes
      versionID: int; // Version number, where 0 is the initial
      typeDefinition: string;
         // Name of the type assembly containing executable code
      versionDiffs: list(STEvolutionStep);
         // Evolution steps to migrate from versionID-1
End_Data_Structure TypeVersion;
```

A.3.3.2 Reification of simple types: SimpleTypeDescription

```
TypeDescription Aspect SimpleTypeDescription
   using I_SimpleTypeDescription

   Attributes
      Constant
         typeName: string;
         kind: string; // Component or Connector
      Variable
         // Data structures for type reification
         currentVersion: int;
         aspects: list(AspectInfo);
         weavings: list(WeavingInfo);
         ports: list(PortInfo);
         constructors: list(ConstructorInfo);
         destructor: list(DestructorInfo);

         // Auxiliary attributes
         currentTypeVersionFile: string;
            // File with the binary code of the current version
         typeSpecObject: SimpleSpec;
            // Keeps the last generated version of SimpleSpec
         reificationIsValid: boolean;
            // True if data stored about the type is still valid
         reificationsBlocked: boolean;
            // If true, reify operations are temporarily suspended
         oldVersions: list(TypeVersion);
            // Version management

   Services
      in begin(typeName: string,kind: string)
         Valuations
            [begin(typeName,kind]
            this.typeName=typeName;
            this.kind=kind;

      in Reify(output reification: SimpleSpec)
         // Ommitted...

      in Stop()       // Blocks reifications temporarily
         Valuations
            [Stop()] reificationsBlocked=true;

      in VersionChanged(newTypeDefinition:string,
               versionDiffs: list(STEvolutionStep))
         Valuations
            [VersionChanged(newTypeDefinition,versionDiffs)]
            currentTypeVersionFile=newTypeDefinition;
            oldVersions.add(new TypeVersion(currentVersion+1,
               newTypeDefinition, versionDiffs);
            reificationIsValid=false;
            reificationsBlocked=false;

      in GetIncrementalChanges(sourceVersion: int,
               targetVersion: int,
               output versionDiffs: list(STEvolutionStep));

      // Services offered through the TypeIntrospectionPort
      in getPorts(output portsList: list);
      in getAspectTypes(output aspectTypesList: list);
      in getWeavings(output weavingList: list);
      in getAspectTypeProperties(aspectName: string,
            output typeDefinition: string, output concern: string,
            output interfaces: list, output playedRoles: list);
      in getPortProperties(portName: string,
            output isProvided: boolean, output isRequired: boolean,
            output interface: string, output playedRole: string);
      in getWeavingProperties(id: string,
            output sourceAspect: string, output sourceMethod: string,
            output sourceParameters: list, output weavingType: string,
            output targetAspect: string, output targetMethod: string,
         output targetParameters: list, output transfFunctions: list);
      ...
...
End TypeDescription Aspect SimpleTypeDescription;
```

A.3.3.3 Reification of composite types: CompositeTypeDescription

```
TypeDescription Aspect CompositeTypeDescription
   using I_CompositeTypeDescription

   Attributes
      Constant
         typeName: string;

      Variable
         // Data structures for type reification
         currentVersion: int;
         systemPorts: list(PortInfo);
         archElements: list(AETypeInfo);
         attachments: list(AttTypeInfo);
         bindings: list(BindTypeInfo);
         constructors: list(ConstructorInfo);
         destructor: list(DestructorInfo);

         // Auxiliary attributes
         currentTypeVersionFile: string;
            // File with the binary code of the current version
         typeSpecObject: CompositeSpec;
            // Keeps the last generated version of CompositeSpec
         reificationIsValid: boolean;
            // True if data stored about the type is still valid
         reificationsBlocked: boolean;
            // If true, reify operations are temporarily suspended
         oldVersions: list(TypeVersion);
            // Version management

   Services
      in begin(typeName: string)
         Valuations
            [begin(typeName] this.typeName=typeName;

      in Reify(output reification: CompositeSpec)
         // Ommitted...

      in Stop()       // Blocks reifications temporarily
         Valuations
            [Stop()] reificationsBlocked=true;

      in VersionChanged(newTypeDefinition:string,
               versionDiffs: list(CTEvolutionStep))
         Valuations
            [VersionChanged(newTypeDefinition,versionDiffs)]
            currentTypeVersionFile=newTypeDefinition;
            oldVersions.add(new TypeVersion(currentVersion+1,
               newTypeDefinition, versionDiffs);
            reificationIsValid=false;
            reificationsBlocked=false;

      in GetIncrementalChanges(sourceVersion: int,
               targetVersion: int,
               output versionDiffs: list(CTEvolutionStep));

      // Services offered through the TypeIntrospectionPort
      in getTypeSpecification(output xmlSpec: string)
         Valuations
         [getTypeSpecification(xmlSpec)]
         xmlSpec=BuildXMLSystemTypeSpec();

      in getTypeName(output typeName: string)
         Valuations
         [getTypeName(typeName)] typeNAme=this.typeName;

      in getSystemPorts(output portsList: list)
         Valuations
         [getSystemPorts(portsList)] portsList=systemPorts;

      in getArchElementTypes(output archElementList: list)
         Valuations
         [getArchElementTypes(archElementList)]
         archElementList=archElements;

      in getAttachmentTypes(output attachmentList: list)
         Valuations
         [getAttachmentTypes(attachmentList)]
         attachmentList=attachments;

      in getBindingTypes(output bindingList: list)
         Valuations
         [getBindingTypes(bindingList)]
         bindingList=bindings;

      in getArchTypeProperties(aeName: string,
            output typeDefinition: string, output portList: string,
            output minCard: integer, output maxCard: integer);
         Valuations
         archElements.Contains(aeName)
         [getArchTypeProperties(...)]
            i=archElements.IndexOf(aeName);
            typeDefinition=archElements[i].typeRef;
            portList=archElements[i].typeRef.Ports();
            minCard= archElements[i].minCard;
            maxCard= archElements[i].maxCard;

      in getPortProperties(archElemType: string, portName: string,
            output isProvided: boolean, output isRequired: boolean,
            output interface: string)
         Valuations
            ... // ommitted

      in getAttachmentTypeProperties(attType: string,
            output srcArchElemType: string, output srcAEport: string,
            output srcMinCard: integer, output srcMaxCard: integer,
            output trgArchElemType: string, output trgAEport: string,
            output trgMinCard: integer, output trgMaxCard: integer)
         Valuations
            ... // ommitted
      in getBindingTypeProperties(bindType: string,
            output systemPortName: string,
            output trgArchElemType: string, output trgAEport: string,
            output trgMinCard: integer, output trgMaxCard: integer)
         Valuations
            ... // ommitted

      in getConnectionsofType(archElemType: string,
            output attachments: list, output bindings)
         Valuations
            ... // ommitted

      in getAttachedTypes(archElemType: string,
            output attachedArchTypes: list)
         Valuations
            ... // ommitted

   External Functions
      BuildXMLSystemTypeSpec();
         // Builds a PRISMA XML specification from current type spec.
         // This XML also includes the type version number.
   End_External Functions

End TypeDescription Aspect CompositeTypeDescription;
```

A.3.4 Type Evolution Aspect

The following fragment shows the specification of a type evolution aspect for simple types. In case of composite types, the difference is that the data structures *CompositeSpec* and *CTEvolutionStep* are used instead of *SimpleSpec* and *STEvolutionStep*, respectively.

```
TypeEvolution Aspect SimpleTypeEvolution

   Attributes
      Variable
         // Auxiliary variables
         isEvolving: boolean; // A type version is being generated

         // Code generation templates
         builderGenerationTemplate: string;

   Services
      in begin(codeTemplates: string)
         Valuations
            [begin(codeTemplates]
            builderGenerationTemplate=codeTemplates;
            isEvolving=false;

      in Reflect(specification: SimpleSpec,
                  evolParams: EvolutionPolicy);
         Valuations
            // Ommitted...

      out NewVersionGenerated(versionID: int, typeDefFile: string,
            versionDiffs: list(STEvolutionStep),
            evolPolicy: EvolutionPolicy);
```

```
...
End TypeEvolution Aspect SimpleTypeEvolution;
```

A.3.5 Evolution Monitoring Aspect

A.3.5.1 Data structures

InstanceInfo

```
Data structure InstanceInfo
   Attributes
      ID: string;
      reference: string;      // Pointer to the instance
      currentVersion: int;
End_Data_Structure InstanceInfo;
```

EvolutionProcess

```
Data structure EvolutionProcess
   Attributes
      targetVersion: int;
      instancesToEvolve: list(InstanceInfo);
      remainingRetries: int;
      evolutionPolicy: EvolutionPolicy;
End_Data_Structure PendingEvolution;
```

EvolutionPolicy

```
Data structure EvolutionPolicy
   Attributes
      // Strategy for evolving instances
      evolutionStrategy : { OnlyNew | EvolveAll | ExcludeSome };
      exclusionSet : list(string);

      // Maximum time each instance has to apply the changes
      evolutionTimeoutPerInstance : int; // in milliseconds
      evolutionRetriesIfTimeoutExceeded : int;
      actionIfRetriesExceeded: { ForceEvolution | AbortEvolution };

   // Default evolution policy: evolve all instances in 1 minute
   // Otherwise, abort the evolution of the instance
   new() {
      evolutionPolicy = EvolveAll;
      exclusionSet = list[];
      evolutionTimeoutPerInstance = 10000;
      evolutionRetriesIfTimeoutExceeded = 6;
      // Maximum total time to evolve: 10000x6=60000 milliseconds
      actionIfRetriesExceeded = AbortEvolution;
   }
End_Data_Structure EvolutionPolicy;
```

A.3.5.2 Simple components: SimpleInstanceMonitoring

The following fragment shows the specification of an evolution monitoring aspect for simple types. In case of composite types, the only difference is that the data structure *CTEvolutionStep* is used instead of *STEvolutionStep*.

```
EvolutionMonitoring Aspect SimpleInstancesMonitoring
```

```
    Attributes
       Variable
          // Instance population
          population: list(InstanceInfo);
          // List of time-constrained active evolution processes
          activeEvolutionProcesses: list(EvolutionProcess);

    Services
       in ReflectToInstances(evolutionSteps: list(STEvolutionStep),
             evolutionParams: EvolutionPolicy);
          Valuations
             // Ommitted...

       EvolutionProcessMonitoring(evProcess: EvolutionProcess);
          // This private service is executed periodically to
          // supervise the evolution of instances
       suspend(timeout : integer);
          // This function suspends the current process

       in RegisterInstance(instanceRef: InstanceInfo)
          Valuations
          [RegisterInstance(instanceRef)]
          population.add(instanceRef);

       in UnRegisterInstance(instanceRef: InstanceInfo)
          Valuations
          [UnRegisterInstance(instanceRef)]
          population.remove(instanceRef);
       ...
 ...
 End EvolutionMonitoring Aspect SimpleInstancesMonitoring;
```

A.3.6 Builder Aspect

A.3.6.1 Example for simple types: ImageProcCardBuilder

The following code shows how a Builder aspect of a simple type, the *ImageProcCardBuilder*, looks like. This code is platform-dependent, because it must be directly executable.

```
public class ImageProcCardBuilder_v0 : IBuilder {
   private int typeVersion = 0;
   public int GetVersion { get {return typeVersion;}}

   SimpleSpec specification;
   public SimpleSpec Specification { get {return specification;}}

   public ImageProcCardBuilder_v0() {
      // Definition of relevant type information
      this.specification = new SimpleSpec(typeof(ImageProcCard),
         "AgroBot", AElementType.Component);
      specification.defineConstructor(
         new ParamInfo[] {
            new ParamInfo("cameraPosition", typeof(string))
         },
         new ExpressionSpec[] {
            new ExpressionSpec(
               "ImageProcCardController.begin(cameraPosition);"),
            new ExpressionSpec("ImageProcCardGUI.begin();")
         }
      )
      specification.defineDestructor(
         new ExpressionSpec[] {
            new ExpressionSpec("ImageProcCardGUI.end();"),
            new ExpressionSpec("ImageProcCardController.end();")
         }
      )
   }

   public void BuildInstance(ImageProcCard comp,
                  string cameraPosition)
   {
      // Building and initialisation of instances
      IAspect aspect1= new ImageProcCardController(cameraPosition);
      comp.AddAspect(aspect1);

      IAspect aspect2= new ImageProcCardGUI();
      comp.AddAspect(aspect2);

      comp.AddWeaving(aspect2, "showImage", "image",
         WeavingType.AFTER, aspect1, "newProcessedImage", "image");

      comp.AddPort("VideoIn", "I_VideoServices", "VIDEOCARD");
      comp.AddPort("ImageOut", "I_ImageProcessingServices",
         "IMAGEANALYZER");
   }

   public void DestroyInstance(ImageProcCard comp) {
      comp.GetAspect(typeOf(ImageProcCardGUI)).Dispose();
      comp.GetAspect(typeOf(ImageProcCardController)).Dispose();
      comp.Dispose();
   }
}
```

A.3.6.2 Code generation pattern for simple types

This is the code generation pattern, in C#, which is used to dynamically generate a *Builder* aspect for simple types.

```
using System.Reflection;
using PRISMA;
using PRISMA.Aspects;
using PRISMA.Aspects.Types;
using PRISMA.Components;
using PRISMA.Components.Weavings;
using PRISMA.Middleware;

[assembly:AssemblyVersionAttribute("%__Version__%")]
namespace %__namespace__% {

public class %__TypeName__%Builder_v%__Version__% : IBuilder {
   private int typeVersion = %__Version__%;
   public int GetVersion { get {return typeVersion;}}

   SimpleSpec specification;
   public SimpleSpec Specification { get {return specification;}}
   public %__TypeName__%Builder_v%__Version__%() {
      // Definition of relevant type information
      this.specification = new SimpleSpec(
         (typeof(%__TypeName__%), "%__namespace__%",
         %__AEType__%);
      // Addition of meta-information to the SimpleSpec object
      %__Specification__%
   }

   public void BuildInstance(IComponent comp,
         %__ConstructorParametersDefinition__%)
   {
      // Building and initialisation of instances
      %__ComponentCreation__%

   }

   public void DestroyInstance(%__TypeName__% comp) {
      %__ComponentDestruction__%
      comp.Dispose();
   }
}}
```

A.3.7 Instance Evolution Planning aspect

A.3.7.1 Data structure: EvolutionRequest

```
Data structure EvolutionRequest
   Attributes
      sourceVersion: integer, derivation: targetVersion-1;
      targetVersion: integer;
      evolutionSteps: list(STEvolutionStep);
      evolutionFlags: { forceEvolution | abortEvolution };
End_Data_Structure EvolutionRequest;
```

A.3.7.2 Aspect InstanceEvolutionPlanning

```
InstanceEvolutionPlanning Aspect
...

Attributes
   Constant
      instanceID: string;

   Variable
      instance_version: int;  // Current version
      pendingEvolutions: queue(EvolutionRequest);
         // list of pending evolutions: versionID and set of changes
      isEvolving: boolean;
         // If the instance is evolving to another version.

      // Auxiliary variables for evolution process management
      newVersion: int;
      evolutionSteps: list(STEvolutionStep);
      partsToStop: list(string);
      partsToStart: list(string);

      // Variables for transaction management
      transactionID: string;
      transState: string = "COMMITTED"; // Default value
      aspectsAdded : list;
      aspectsToRemove : list;
      weavingsToRemove : list;
      weavingsAdded : list;
      portsToRemove : list;
      portsAdded : list;

Triggers
   // Initiation of an evolution process
   TransformInstance() when
      pendingEvolutions.Size()>0 && isEvolving=false;

Services
   // *** Type-level Interaction Services ***
   in ReflectToInstance(newVersion: int,
         evolutionSteps: list(STEvolutionStep))
      Valuations
         [ReflectToInstance(newVersion, evolSteps)]
         pendingEvolutions.Add( new EvolutionRequest(
            targetVersion=newVersion, evolutionSteps=evolSteps));

   in ForceEvolution(versionID: int);
   in AbortEvolution(versionID: int);

   in GetCurrentVersion(output versionID: int);
   out GetIncrementalChanges(currentVersion: int, targetVersion: int,
         output evolutionSteps: list(STEvolutionStep));

   // *** Internal Evolution Services ***
   TransformInstance()
      Valuations
         [TransformInstance()]
         // Processing of a pending evolution request
         evolutionRequest = pendingEvolutions.dequeue();
         newVersion = evolutionRequest.targetVersion;
         evolutionSteps = evolutionRequest.evolutionSteps;

   // *** Atomic change operations ***
   AddAspect(aspectTypeName: string, parameters: string);
   RemoveAspect(aspectTypeName: string);
   ReplaceAspect(aspectToReplace: string, newAspectType: string,
      newAspectParameters: string);
   AddPort(portName: string, interface: string, playedRole: string);
   RemovePort(portName: string);
   AddWeaving(sourceAspect: string, sourceMethod: string,
      sourceParameters: list, weavingType: string,
      targetAspect: string, targetMethod: string,
      targetParameters: list, transfFunctions: list);
   RemoveWeaving(sourceAspect: string, sourceMethod: string,
      weavingType: string, targetMethod: string);

   // *** Transaction Management ***
   BeginEvolutionTransaction();
      Valuations
         [BeginEvolutionTransaction()]
            // Initialization of transaction management structures
         transState = "ACTIVE";
         aspectsAdded = new list[];
         aspectsToRemove = new list[];
         weavingsToRemove = new list[];
         weavingsAdded = new list[];
         portsToRemove = new list[];
         portsAdded = new list[];
   EndEvolutionTransaction()
      Valuations
         [EndEvolutionTransaction ()] transState="COMMITTED";
   RollbackEvolutionTransaction()
      Valuations
         [RollbackEvolutionTransaction()] transState="ROLLBACKED";

   // *** Starting and Stopping services ***
   out StartAspect(aspectID: string);
   out StopAspect(aspectID: string);
   out StartPort(portName: string);
   out StopPort(portName: string);
   out StopWeaving(sourceAspect: string, targetAspect: string);
   out StartWeaving(sourceAspect: string, targetAspect: string);
...
End_InstanceEvolutionPlanning_Aspect
```

A.3.8 Instance Monitoring aspect

```
InstanceMonitoring Aspect
   ...
   Services
      in getAspects(output aspectIDs: list);
      in getPorts(output portIDs: list);
      in getWeavings(output weavingIDs: list);

      in getAspectProperties(aspectID: string,
            output aspect: object, output linkedPorts: list,
            output definedWeavings: list);
      in getPortProperties(portID: string, output port: object,
            output linkedAspect: string);
      in getWeavingProperties(weavingID:string,
            output sourceAspectID: string,
            output weavingType: string,
            output targetAspectID: string);

      in getStatus(elementID: string, output status: string);
      in getElementsOfStatus(status: string,
            output elemIDList: list);
   ...
End_InstanceMonitoring_Aspect;
```

A.3.9 Instance Effector aspect

```
InstanceEffector Aspect
   ...
   Services
      // *** Services for Safe Stopping ***
      in StopPart(elemID: string);    // Reach a Quiescent status.
      in StartPart(elemID: string);    // Reach an Active status.

      // *** Atomic change operations ***
      in CreateAspect(aspectTypeName: string, initParams: string,
            output aspectID: string);
      in DestroyAspect(aspectID: string);
      in CreatePort(name: string, interface: string,
            playedRole: string, output portID: string);
      in DestroyPort(portID: string);
      in CreateWeaving(sourceAspectID:string, sourceMethod:string,
            sourceParameters: list, weavingType: string,
            targetAspectID: string, targetMethod: string,
            targetParameters: list, transfFunctions: list,
            output weavingID: string);
      in DestroyWeaving(weavingID: string);

      // Services for aspect replacements
      in SerializeAspectState(aspectID: string,
            output state: string);
      in UpdateStateStructure(oldAspectType: string,
            oldState: string, newAspectType: string,
            newRequiredValues: list,
            output transformedState: string);
      in UnserializeAspectState(aspectType: string,
            serializedState: string, output aspect: object);
   ...
End_InstanceEffector_Aspect;
```

APPENDIX B

EXTENSIONS OF THE PRISMA AOADL

This appendix presents some language constructs that have been added to the PRISMA Aspect-Oriented Architecture Description Language (AOADL), in order to support the features introduced in this thesis. These language constructs are: lists (section B.1), iterations and loops (section B.2), and partial definitions (section B.3).

B.1 Lists

A new datatype has been added to the PRISMA ADL: `list`. This datatype defines a variable collection of elements of any PRISMA datatype. This collection is untyped, so it can contain different kind of elements (*integers*, *strings*, *datetimes*, etc.).

The list datatype provides several methods for its manipulation:

- `new list[]`: Creates an empty list.
- `new list[elem1,elem2,...]`: Creates a list which is initialised with the elements *elem1*, *elem2*, ...
- `Size()`: Provides the number of elements that the list contains.
- `Add(element)`: Adds *element* to the list. The list allows duplicated elements.
- `Remove(element)`: Remove the first occurrence of *element* from the list. If *element* is not in the list, this service does nothing.
- `list[i]`: Returns the element that is in the position *i*. The range of accepted values of *i* are: `0<=i<Size`.
- `list[name]` : Returns the element that is identified by “name”.

- `Contains(element)`: Returns true if *element* is contained in the list.
- `IndexOf(element)`: Returns the position of the first item which equals *element.*

In this way, complex data structures (queues, stacks and specific collections) can be built from this base datatype.

B.2 Iterations and Loops

Iterative structures are required to operate on groups of elements, as stated in (Bradbury et al., 2004). These constructs take the form of well known programming structures (*while*, *for* and *foreach*). The modelling of their behaviour, using the PRISMA dialect of poliadic π-calculus (Perez, 2006, pp. 120-126) is provided below:

```
while{COND}(P) ::=
      LOOP ::= if COND then (P --> LOOP) else 0    =>
      *(if COND then P else 0)                     =>
      *({COND}P)
```

```
for(INIT, COND, INCR) (P) ::=
      INIT --> *(if COND then (P --> INCR) else 0)  =>
      INIT --> *({COND}(P --> INCR))
```

```
foreach VAR in LIST do (P) ::=
      <i=0> -->
      *(if i<LIST.Size()
         then ( <VAR=LIST[i]> --> P --> <i=i+1> )
         else 0
      )
```

Where:

- `COND` is the condition used to finish an iteration, e.g. `{i<=5}`
- `P` is a process, as defined in pi-calculus, which is going to be repeated several times.
- `0` is a void process. It means the finalization of the current process.
- `INIT` is a process which defines a variable and sets an initial value, e.g. `<i=0>`

- `INCR` is a process which increases the variable defined in `INIT`, eg. `<i=i+1>`
- `LIST` represents a list or collection of items (variables), eg. `list={"e1","e2","e3"}`. It provides a method `Size()` for retrieving the number of items, and an accessor to get its elements by means of an index, e.g. `list[i]`.

These structures are useful for specifying configuration transactions: for iterating on the list of existing instances and select one in particular if some conditions apply. For instance, the following PRISMA specification fragment uses the *foreach* construct for selecting, among the set of instances attached to a given instance (*imageProcCard-ID*), only those that are of type *VCC-Conn* or *IPC-Conn*.

```
AttachedElements!(ImageProcCard-ID, output attachedIDs) -->
foreach elem in attachedIDs do (
      ( if TypeOf(elem)=="VCC-Conn" then ...)
   +  ( if TypeOf(elem)=="IPC-Conn" then ...)
)
```

Another example of the use of iterations is shown in the fragment below, which belongs to the *IncrementalStart* configuration transaction (defined in the VisionSystemReconfigurationAnalysis aspect, see A.2.3). In this fragment, a *replace* action is executed for each instance of *imageProcSoftware* component:

```
...
   foreach elem in imageProcSoftware-list do (
      replace-ImageProcSoftware!(elem, cameraPos, output newID);
   ) -->
...
```

In this way, PRISMA reconfiguration specifications are simplified and easier to understand for architects unfamiliar with process algebras and pi-calculus in particular.

B.3 Partial Definitions of Software Artefacts

The framework presented in this thesis has introduced several software artefacts (e.g. the *Evolver* component, the *Reconfiguration Analysis* aspect) that have user-defined sections and automatically-generated sections. The automatically-generated sections define the synchronizations with other elements of the evolution infrastructure, and are regenerated each time the type is changed. This is the case of the Reconfiguration Analysis aspect, which domain-specific services can change if the composite type is changed (see section 6.5.4.2, page 236). The user-defined sections must be defined separately, to avoid that the compiler may remove them when regenerating the specifications again, thus losing user-defined functionality. This is an issue that is common in most of model-driven development supporting tools.

To deal with this issue, the concept of **partially defined software artefacts** has been used:

> *A partially-defined artefact is a software artefact (i.e. a type, a class, a specification) that is splitted in several parts, which are combined in the compiling process.*

This concept is particularly useful to combine user-defined specifications with automatically-generated specifications, while also maintaining them separated for code-generation purposes. This concept has been integrated into the PRISMA AOADL in the following way:

- Software artefacts that can be partially defined are: *aspects*, *components*, *connectors* and *systems*.
- A software artefact is partially defined by a base specification and a set of partial specifications (one or more).
  - The base specification defines almost all the sections of the software artefact, but leaves some undefined. This specification uses the keyword:

    is partially defined by <set of partial specifications>
  - Partial specifications extend the base specification by adding more elements to an already defined section (e.g. adding more ports to the ports section defined in the base specification of a component), or by adding elements to a section that has not been defined (e.g. the base specification of a component does not defines weavings). This specification uses the keyword:

    is partial

Next, it is shown how the PRISMA AOADL syntax has been extended to include these special keywords[52]:

[52] It is only shown the fragment of the PRISMA AOADL syntax where the partial concept is used. For more details about the complete PRISMA AOADL, see (Pérez, 2006)

```
<aspect> ::= <concern> Aspect <aspect_name>
               [is partially defined by
                      <list_aspect_names_of_the_same_concern> |
                is partial ]
               [using <interface_name_list>]
               ...

<component> ::= Component <component_name>
                  [is partially defined by <list_of_component_name > |
                   is partial ]
                  <aspects_importation_seq>
                  ...

<connector> ::= Connector <connector_name>
                  [is partially defined by <list_of_connector_name > |
                   is partial ]
                  <aspects_importation_seq>
                  ...

<system> ::= System <system_name>
                  [is partially defined by <list_of_system_name > |
                   is partial ]
                  <aspects_importation_seq>
                  ...
```

Examples of how the *partial* concept is used in PRISMA AOADL specifications are provided in Appendix A, sections A.2.2, A.2.3, and A.2.4, so are not included here.

LIST OF FIGURES

Figure 2.1. *Advantages for Modularity with AOP* 45
Figure 2.2. *Crosscutting-concerns in PRISMA architectures* 49
Figure 2.3. *Black-box view of PRISMA architectural elements* 51
Figure 2.4. *Internal view of simple PRISMA elements* 52
Figure 2.5. *Internal view of composite PRISMA elements* 52
Figure 2.6. *Example of a composite type and two possible instantiations* 54
Figure 2.7. *Example of a PRISMA System* 54
Figure 2.8. *Example of PRISMA Configurations* 55
Figure 2.9. *Meta-Object Facility layers and PRISMA models* 56
Figure 2.10. *MDD from the PRISMA Metamodel to Applications* 58
Figure 2.11. *PRISMA CASE (Perez, 2006)* 59
Figure 3.1. *A Feedback and Feedforward control system (Morrison et al., 2007)* 85
Figure 3.2. *Internal Structure of an Autonomic Manager (Tewari & Milenkovic, 2006)* 87
Figure 3.3. *Reference architecture for Autonomic Computing (IBM, 2006)* 89
Figure 3.4. *Computational reflection: main concepts* 91
Figure 5.1. *Agricultural robot designs* 137
Figure 5.2. *Agricultural robot prototypes* 137
Figure 5.3. *Software architecture of the Agrobot* 140
Figure 5.4. *Notation used* 140
Figure 5.5. *VisionSystem architectural type* 143
Figure 5.6. *PRISMA specification of the VisionSystem type* 144
Figure 5.7. *PRISMA specification of RightCamera and LeftCamera instances* 145
Figure 5.8. *White-box view of the RightCamera component instance* 145
Figure 5.9. *PRISMA specification of the ImageProcCard component* 146

Figure 5.10. PRISMA specification of the ImagProcSwController aspect..............148
Figure 5.11. VisionSystem type with fault-tolerance support..................................153
Figure 5.12. Example of alternative configuration in case of failure 153
Figure 6.1. Aspects for autonomic reconfiguration...165
Figure 6.2. Example of weavings among aspects..167
Figure 6.3. Services of the Monitoring aspect..169
Figure 6.4. Example of PRISMA XML Configuration Model..............................171
Figure 6.5. Event interception behaviour...176
Figure 6.6. Pattern of the Reconfiguration Analysis Aspect...................................180
Figure 6.7. Example of a Reconfiguration Analysis Aspect (fragment)..................182
Figure 6.8. Reconfiguration triggers: activation by attributes.................................183
Figure 6.9. Reconfiguration triggers: activation by service invocations...................184
Figure 6.10. Reconfiguration triggers: activation by event interceptions 185
Figure 6.11. Example of Configuration Transactions:
RepairImageProcessingUnit...186
Figure 6.12. Example of Configuration Transactions: IncrementalStart................188
Figure 6.13. Example of auxiliary attributes provided by the Rec.Analysis aspect..189
Figure 6.14. Example of domain-specific reconfiguration services 194
Figure 6.15. PRISMA specification of BeginConfigurationTransaction.................198
Figure 6.16. PRISMA specification of EndConfigurationTransaction...................200
Figure 6.17. PRISMA specification of CheckArchitectureConsistency 200
Figure 6.18. PRISMA specification of RollbackConfigurationTransaction............202
Figure 6.19. Protocol section of the VisionSystemReconfigurationServices aspect...203
Figure 6.20. PRISMA specification of CreateArchElem..205
Figure 6.21. PRISMA specification of CreateAttachment & CreateBinding.........207
Figure 6.22. PRISMA specification of DestroyArchElem 208
Figure 6.23. PRISMA specification of DestroyAttachment & DestroyBinding 210
Figure 6.24. PRISMA specification of ReplaceArchElem......................................213
Figure 6.25. Services of the Reconfiguration Effector aspect...................................215
Figure 6.26. Required Interfaces for Supporting Dynamic Updating......................219
Figure 6.27. Architecture of the VisionSystem type including
an Evolver component 223
Figure 6.28. PRISMA specification of the VisionSystem Type 224
Figure 6.29. Internal Structure of an Evolver component......................................225

Figure 6.30. *Extension of the PRISMA metamodel with the Evolver Component* .226
Figure 6.31. *Default Introspection Interface for the VisionSystemEvolver*.............227
Figure 6.32. *Default Reconfiguration Interface for the VisionSystemEvolver*.........229
Figure 6.33. *Example of declaration of out services of a Reconf. Analysis aspect*...230
Figure 6.34. *Example of weaving definitions in the Evolver component*................232
Figure 6.35. *Metamodel of the Evolver Component*...234
Figure 6.36. *Template for Evolver component types*...235
Figure 6.37. *Fragment of VisionSystemEvolverMechanisms*.................................238
Figure 6.38. *Architecture of the RightCamera composite instance, with the detail of the reconfiguration aspects provided by an Evolver instance*..........240
Figure 6.39. *Sequence of interactions among reconfiguration aspects as a result of a configuration transaction*...242
Figure 6.40. *Architecture of the RightCamera composite instance, after the execution of the RepairImageProcessingUnit reconfiguration process*................243
Figure 7.1. *Reflective Evolution of Architectural Types*...257
Figure 7.2. *Model of Type Meta-Instances*...258
Figure 7.3. *Structure of a SimpleSpec object*...260
Figure 7.4. *Black-box view of the dynamic evolution process*.................................262
Figure 7.5. *Capturing evolution over time*..267
Figure 7.6. *Synchronous evolution model*...269
Figure 7.7. *Asynchronous evolution model*...270
Figure 7.8. *Internal structure of a Type Meta-Instance*..275
Figure 7.9. *Weavings that manage the population of instances*.............................275
Figure 7.10. *Example of an automatically generated evolvable type: ImageProcCard*...277
Figure 7.11. *Example of an automatically generated Builder aspect: ImageProcCardBuilder*...279
Figure 7.12. *Fragment of the TypeDescription aspect of simple arch. elements*......282
Figure 7.13. *Introspection services provided by a SimpleTypeDescription aspect*....283
Figure 7.14. *Fragment of the TypeEvolution aspect of simple arch. elements*.........286
Figure 7.15. *Weavings used to temporarily block other meta-instance concerns*......287
Figure 7.16. *Code generation pattern for Builder aspects of simple types*...............290
Figure 7.17. *Fragment of the code generation process for aspects in C#*................290
Figure 7.18. *Weavings activated when the Reflect service finishes*..........................292

Figure 7.19. *Fragment of the EvolutionMonitoring aspect of simple architectural elements*........293
Figure 7.20. *Data structure STEvolutionStep*........295
Figure 7.21. *EvolutionPolicy data structure*........296
Figure 7.22. *EvolutionProcess data structure*........298
Figure 7.23. *Integration of instance-level evolution aspects in simple instances*......301
Figure 7.24. *Services of the InstanceEvolutionPlanning aspect*........303
Figure 7.25. *Services provided by the InstanceMonitoring aspect*........310
Figure 7.26. *Services provided by the InstanceEffector aspect*........312
Figure 7.27. *Sequence of evolution instructions performed to dynamically update the type ImageProcSoftware*........315
Figure 8.1. *Evolution operations over Composite types*........323
Figure 8.2. *Example of two evolution processes*........324
Figure 8.3. *Graphical specification of a type tagged with Evolution tags*........325
Figure 8.4. *Textual specification of a type tagged with Evolution tags*........326
Figure 8.5. *Function GetTypeVersionSpecification*........326
Figure 8.6. *Rule R3: AddArchitecturalType*........329
Figure 8.7. *Rule R20 - CreateArchitecturalElement*........332
Figure 8.8. *Rule R21 - ActivateInstantiatedArchElements*........333
Figure 8.9. *Rule R38 - AdvanceInstanceToNextVersion*........335
Figure 8.10. *Rule R5 - UpdateAEType*........338
Figure 8.11. *Rule R37 - ReplaceAE*........340
Figure 8.12. *Type Graph of PRISMA with Evolution Tags*........344
Figure 8.13. *Instance graph of the Sys type (at version 1) with the C1 instance (at version 0)*........346
Figure 8.14. *CreateAE: mapping of rule CreateAE in the AGG tool*........347
Figure 8.15. *NACs of the graph transformation rule CreateAE*........348
Figure 8.16. *Attribute conditions of the graph transformation rule CreateAE*.......349

REFERENCES

(Abowd et al., 1993) G.D. Abowd, R. Allen, D. Garlan. *Using style to understand descriptions of software architectures*. ACM Software Engineering Notes, 18(5):9–20, 1993.

(Adamek & Plasil, 2005) J. Adamek, F. Plasil. *Component composition errors and update atomicity: static analysis*. Journal of Software Maintenance and Evolution, 17:363-377. Wiley, 2005

(AGG, 2010) TU Berlin Graph Grammar Group. AGG: *Attributed Graph Grammar System Tool.* In: http://user.cs.tu-berlin.de/~gragra/agg/

(Aksit et al., 1994) M. Aksit, K. Wakita, J. Bosch, L. Bergmans, Y. Yonezawa. *Abstracting Object Interactions Using Composition Filters*. In: ECOOP'93, workshop on Object-Based Distributed Programming, 1994.

(Aldrich et al., 2002) J. Aldrich, C. Chambers, D. Notkin. *ArchJava: Connecting Software Architecture to Implementation*. International Conference on Software Engineering (ICSE'02). Orlando, USA, 2002.

(Ali et al., 2006) N. Ali, J. Pérez, C. Costa, I. Ramos, J.A. Carsí. *Mobile Ambients in Aspect-Oriented Software Architectures*. K. Sacha (ed.): Software Engineering Techniques: Design for Quality. IFIP Series, vol. 227, pp. 37-48. Springer, 2006.

(Ali, 2008) N. Ali. *Ambients in an Aspect-Oriented Software Architecture*. PhD Thesis, Universidad Politécnica de Valencia, February 2008.

(Aliaga-Varea, 2008) S. Aliaga-Varea. *Dynamic Reconfiguration of Aspect-Oriented Software Architectures using the .NET platform*. Ms Science Thesis. Faculty of Computer Science, Universidad Politécnica de Valencia, September 2008 (in Spanish).

(Allen & Garlan, 1997) R. Allen, D. Garlan. *A formal basis for architectural connection*. ACM Transactions on Software Engineering and Methodology. July, 1997.

(Allen et al., 1998) R. Allen, R. Douence, D. Garlan. *Specifying and Analyzing Dynamic Software Architectures*. Fundamental Approaches to Software Engineering (FASE'98). LNCS, vol. 1382, pp. 21–37. Springer, 1998.

(Almeida et al., 2001) J.P. Almeida, M. Wegdam, L. Pires, and M. van Sinderen. *An approach to dynamic reconfiguration of distributed systems based on object-middleware*. 19th Brazilian Symposium on Computer Networks (SBRC 2001). Santa Catarina, Brazil, May 2001.

(Andersson & Ritzau, 2000) J. Andersson, T. Ritzau. *Dynamic code update in Jdrums*. Workshop on Software Engineering for Wearable and Pervasive Computing. Limerick, Ireland, June 2000.

(Andersson et al., 1998) J. Andersson, M. Comstedt, T. Ritzau. *Run-time Support for Dynamic Java Architectures*. ECOOP'98 workshop on Object-Oriented Software Architectures (WOOSA'98). Brussels, 1998. Available as Technical report 13/98 University of Karlskrona/Ronneby.

(Andersson et al., 2009) J. Andersson, R. De Lemos, S. Malek, D. Weyns. *Reflecting on Self-Adaptive Software Systems*. ICSE workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'09). Vancouver, Canada, 2009.

(Andersson et al., 2009a) J. Andersson, R. De Lemos, S. Malek, D. Weyns. *Modeling Dimensions of Self-Adaptive Software Systems*. Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]. LNCS vol. 5525, pp. 27-47. Springer 2009.

(Andrade & Fiadeiro, 2003) L.F. Andrade, J.L. Fiadeiro. *Architecture Based Evolution of Software Systems*. In M. Bernardo & P. Inverardi (eds.): Formal Methods for Software Architectures. Lecture Notes in Computer Science, vol. 2804. Springer, 2003.

(Apache, 2010) Apache Commons. *Versioning Guidelines*. In: http://commons.apache.org/releases/versioning.html (last accessed on August 2010)

(Aßmann, 2003) U. Aßmann. *Invasive Software Composition*. Springer, 2003.

(Atkinson & Kuhne, 2003) C. Atkinson, T. Kühne. *Model-Driven Development: A Metamodeling Foundation*. Software, 20(5): 36-41. IEEE, 2003.

(Ayed & Berbers, 2007) D. Ayed and Y. Berbers. *Dynamic Adaptation of CORBA Component-Based Applications*. In: ACM Symposium on Applied Computing (SAC'07). Seoul, Korea, March 2007.

(Babaoglu et al., 2005) O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. P. A. van Moorsel. *The Self-Star Vision*. In Self-star Properties in Complex Information Systems, Conceptual and Practical Foundations. Lecture Notes in Computer Science, vol. 3460, pp. 1-20. Springer, 2005.

(Balasubramaniam et al., 2004) D. Balasubramaniam, R. Morrison, K. Mickan, et al. *Support for feedback and change in self-adaptive systems*. ACM SIGSOFT Workshop on Self-Managing Systems (WOSS'04). Newport Beach, CA, USA, 2004.

(Balasubramaniam et al., 2005) D. Balasubramaniam, R. Morrison, G. Kirby, et al. *A software architecture approach for structuring autonomic systems*. Workshop on Design and Evolution of Autonomic Application Software (DEAS'05). St. Louis, Missouri, 2005.

(Barais et al., 2008) O. Barais, A.F. Le Meur, L. Duchien, J. Lawall. *Software Architecture Evolution*. T. Mens, S. Demeyer (eds.): Software Evolution, chapter 10. Springer, 2008.

(Baresi et al., 2004) L. Baresi, R. Heckel, S. Thone, D. Varro. *Style-based refinement of dynamic software architectures*. 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04). June 2004.

(Bas et al., 2003) L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. Second Edition, Addison Wesley, 2003.

(Batista et al., 2005) T. Batista, A. Joolia, G. Coulson. *Managing Dynamic Reconfiguration in Component-Based Systems*. In: 2nd European Workshop on Software Architectures (EWSA'05). LNCS, vol. 3527, pp. 1-17. Springer, 2005.

(Batista et al., 2008) T. Batista, A. Tadeu, G. Coulson, et al. *On the Interplay of Aspects and Dynamic Reconfiguration in a Specification-to-Deployment Environment*. In proc. of: 2nd European Conference on Software Architecture (ECSA'08). LNCS, vol. 5292. Springer, 2008.

(Bencomo et al., 2008) N. Bencomo, G.S. Blair, C.A. Flores-Cortés, P. Sawyer. *Reflective Component-based Technologies to Support Dynamic Variability*. In: 2nd Int. Workshop on Variability Modelling of Software-Intensive Systems (VaMoS'08). Universität Duisburg-Essen, Germany, 2008.

(Bennett & Rajlich, 2000) K.H. Bennett, V.T. Rajlich. *Software maintenance and evolution: a roadmap*. Conference on the Future Of Software Engineering (FOSE'00). Limerick, Ireland, June, 2000.

(Bergadano & Gunetti, 1995) F. Bergadano, D. Gunetti. *Inductive Logic Programming: From Machine Learning to Software Engineering*. MIT Press, December 1995.

(Beydeda et al., 2005) S. Beydeda, M. Book, V. Gruhn. *Model-Driven Software Development*. Springer, 2005.

(Bidan et al., 1998) C. Bidan, V. Issarny, T. Saridakis, A. Zarras. *A dynamic reconfiguration service for CORBA*. IEEE International Conference on Configurable Distributed Systems. May 1998.

(Bierman et al., 2003) G. Bierman, M. Hicks, P. Sewell, G. Stoyle. *Formalizing dynamic software updating*. 2nd InternationalWorkshop on Unanticipated Software Evolution. Warsaw, Poland, April, 2003.

(Blackmore et al., 2004) B.S. Blackmore, S. Fountas, L. Tang, H. Have. *Design specifications for a small autonomous tractor with behavioural control*. Journal of the International Commision of Agricultural and Biosystems Engineering (CIGR). Vol. 6, Manuscript PM 04 001. July 2004.

(Blackmore et al., 2006) B.S. Blackmore, H.W. Griepentrog, S. Fountas. *Autonomous Systems for European Agriculture*. In proc. of Automation Technology for Off-Road Equipment (ATOE). Bonn, Germany, 2006.

(Bloom & Day, 1993) T. Bloom, M. Day. *Reconfiguration and module replacement in Argus: Theory and practice*. Software Engineering Journal, 8(2):102-108, March 1993.

(Boehm, 1981) B.W. Boehm. *A spiral model of software development and enhancement*. IEEE Computer, Vol. 21, No. 5, pp. 61-72, 1988.

(Boronat et al., 2004) A. Boronat, J. Pérez, J.A. Carsí, I. Ramos. *Two Experiences in Software Dynamics*. J. UCS 10(4): 428-453, 2004.

(Bradbury et al., 2004) J.S. Bradbury, J.R. Cordy, J. Dingel, M. Wermelinger. *A Survey of Self-Management in Dynamic Software Architecture Specifications*. In proc. of: First ACM SIGSOFT Workshop on Self-Managed Systems (WOSS'04). Newport Beach, CA, 2004.

(Brosilow & Joseph, 2002) C. Brosilow, B. Joseph. *Techniques of Model-Based Control*. In: Feedforward Control. Prentice-Hall, New York (2002), Chap. 9

(Bruneton et al., 2004) E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, J.B. Stefani. *An open component model and its support in java*. In proc. of the 7th Int. Symposium on Component-Based Software Engineering (CBSE'04). LNCS, Vol. 3054. Springer-Verlag, 2004.

(Buason et al., 2005) G. Buason, N. Bergfeldt, T. Ziemke. *Brains, Bodies, and Beyond: Competitive Co-Evolution of Robot Controllers, Morphologies and Environments*. In: Genetic Programming and Evolvable Machines, vol. 6(1):25-51. March 2005.

(Bucchiarone et al., 2007) A. Bucchiarone, H. Melgratti, S. Gnesi, R. Bruni. *Modelling Dynamic Software Architectures using Typed Graph Grammars*. Graph Transformations for Verification and Concurrency. Electronic Notes on Theoretical Computer Science, Vol. 213, No. 1, pp. 39-53. Elsevier, 2007.

(Buckley et al., 2005) J. Buckley, T. Mens, M. Zenger, A. Rashid, G. Kniesel. *Towards a taxonomy of software change*. Journal of Software Maintenance and Evolution: Research and Practice, 17(5):309-332. Wiley, 2005.

(Bures et al., 2006) T. Bures, P. Hnetynka, F. Plasil. *SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model*. In: 4th International Conference on Software Engineering Research, Management and Applications (SERA'06). USA, 2006.

(Bures et al., 2007) T. Bures, P. Hnetynka, F. Plasil. *Runtime Concepts of Hierarchical Software Components*. International Journal of Computer & Information Science, Vol. 8, No. S, pp. 454-463, 2007.

(Buschmann et al., 1996) F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

(Cámara et al., 2008) J. Cámara, G. Salaün, C. Canal. *Composition and Run-time Adaptation of Mismatching Behavioural Interfaces*. Journal of Universal Computer Science, 14(13):2182-2211. Springer, 2008.

(Canal et al., 1999) C. Canal, E. Pimentel, J.M. Troya. *Specification and Refinement of Dynamic Software Architectures*. In: First Working IFIP Conference on Software Architecture (WICSA'99). San Antonio, Texas, USA, 1999.

(Canal et al., 2006) C. Canal, J.M. Murillo, P. Poizat. *Software Adaptation*. L'Objet, special issue on coordination and adaptation techniques, vol. 12, no. 1, pp. 9-31, 2006.

(Canal et al., 2008) C. Canal, P. Poizat, G. Salaün. *Model-Based Adaptation of Behavioral Mismatching Components*. Transactions on Software Engineering, vol. 34, No. 4. IEEE, 2008.

(Capra & Cazzola, 2009) L. Capra, W. Cazzola. *An Introduction to Reflective Petri Nets*. E.M.O. Abu-Taieh, A.A. El Sheikh (eds.): Handbook of Research on Discrete Event Simulation Environments: Technologies and Applications, chapter 9, pp. 191–217. IGI Global, November 2009.

(Carzaniga et al., 2007) A. Carzaniga, G.P. Picco, G. Vigna. *Is Code Still Moving Around? Looking Back at a Decade of Code Mobility*. In: 29th International Conference on Software Engineering (May 20 - 26, 2007). IEEE Computer Society, Washington DC, 2007.

(Cazzola et al., 1999) W. Cazzola, A. Savigni, A. Sosio, F. Tisato. *Rule-Based Strategic Reflection: Observing and Modifying Behavior at the Architectural Level*. 14th IEEE Int. Conf. on Automated Software Engineering. IEEE, 1999.

(Cazzola et al., 1999a) W. Cazzola, A. Savigni, A. Sosio, F. Tisato. *Architectural Reflection: Concepts, Design, and Evaluation*. Technical Report RI-DSI 234-99, DSI, Università degli Studi di Milano, May 1999.

(Cazzola et al., 2004) W. Cazzola, A. Ghoneim, G. Saake. *Software Evolution through Dynamic Adaptation of Its OO Design*. In: Objects, Agents, and Features. LNCS, vol. 2975, pp. 31-48. Springer, Heidelberg, 2004.

(Cazzola et al., 2007) W. Cazzola, S. Chiba, G. Saake. *Guest Editors' Introduction: Aspects and Software Evolution*. Transactions on Aspect-Oriented Software Development, 4: 114-116. Springer, 2007.

(Cazzola, 1998) W. Cazzola. *Evaluation of Object-Oriented Reflective Models*. ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98), Brussels, Belgium, July 1998.

(Cazzola, 2009) W. Cazzola. *Cogito, Ergo Muto!* In proc. of Workshop on Self-Organizing Architectures (SOAR'09), held at the Joint 8th Working IEEE/IFIP Conference on Software Architecture & 3rd European Conference on Software Architecture (WICSA/ECSA 2009). Cambridge, UK, 14th September 2009.

(Chakravarti et al., 2003) A.J. Chakravarti, X. Wang, J.O. Hallstrom, G. Baumgartner. *Implementation of Strong Mobility for Multi-Threaded Agents in Java.* In: Proc. of International Conference on Parallel Processing (ICPP'03), 2003.

(Chapin et al., 2001) N. Chapin, J.E. Hale, K.M. Kham, J.F. Ramil, W. Tan. *Types of software evolution and software maintenance*. Journal of Software Maintenance, Vol. 13, pp. 3-30, 2001.

(Cheng et al., 2005) S. Cheng, D. Garlan, B.R. Schmerl. *Making Self-Adaptation an Engineering Reality*. Self-star Properties in Complex Information Systems. LNCS, vol. 3460, pp. 158-173. Springer, Heidelberg, 2005.

(Cheng et al., 2009) B.H.C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, et al. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. In Software Engineering for Self-Adaptive Systems, LNCS vol. 5525, pp. 1-26. Springer, 2009.

(Chitchyan & Sommerville, 2004) R. Chitchyan, I. Sommerville. *Comparing Dynamic AO Systems*. In proc. of: Dynamic Aspects Workshop (DAW'04). Lancaster, UK, March 2004.

(Chitchyan et al., 2005) R. Chitchyan, A. Rashid, P. Sawyer et al. *Report Synthesizing State-of-the-Art in Aspect-Oriented Requirements Engineering,* *Architectures and Design*. Technical Report AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9. Lancaster Univ., UK, 2005.

(Clements et al., 2002) P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2002.

(Cockburn, 2001) A. Cockburn. *Agile Software Development*. Addison-Wesley, 2001.

(Cook et al., 2006) S. Cook, R. Harrison, M.M. Lehman, P. Wernick. *Evolution in software systems: foundations of the SPE classification scheme*. Journal of Software Maintenance and Evolution: Research and Practice, Vol. 18, No. 1, pp. 1-35, 2006.

(Costa-Soria & Heckel, 2010) C. Costa-Soria, R. Heckel. *Modelling the Asynchronous Dynamic Evolution of Architectural Types*. Self-Organizing Architectures. Lecture Notes on Computer Science, vol. 6090, pp. 198-229. Springer-Verlag, Berlin Heidelberg, July 2010.

(Costa-Soria et al., 2006) C. Costa, N. Ali, C. Millán, J.A. Carsí. *Transparent Mobility of Distributed Objects using .NET*. In proc. of 4th International Conference on .NET Technologies, pp. 11-18. Pilsen, Czech Republic, June 2006.

(Costa-Soria et al., 2011) C. Costa-Soria, J. Pérez, J.A. Carsí. *An Aspect-Oriented Approach for Supporting Autonomic Reconfiguration of Software Architectures*. Special Issue on Autonomic and Self-Adaptive Systems, Informatica (Slovenia), vol. 35, issue 1, pp. 15-27. February 2011. ISSN 0350-5596.

(Costa-Soria, 2005) C. Costa-Soria. *Study and Implementation of an Aspect-Oriented Component-Based Architecture Model on .NET Technology*. Ms Science Thesis. Faculty of Computer Science, Universidad Politécnica de Valencia, March 2005 (in Spanish).

(Costa-Soria, 2005a) C. Costa, J. Pérez, N. Ali, J.A. Carsí, I. Ramos. *PRISMANET middleware: Dynamic Evolution Support for Aspect-Oriented Software Architectures*. In: X Jornadas de Ingeniería del Software y Bases de Datos (JISBD'05). Granada, September 2005 (in Spanish)

(Coulson et al., 2004) G. Coulson, G.S. Blair, P. Grace, A. Jolia, K. Lee, J. Ueyama. *OpenCOM v2: A Component Model for Building Systems Software*. In: IASTED Software Engineering and Applications. Cambridge (MA), USA, 2004.

(Coulson et al., 2008) G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, T. Sivaharan. *A generic component model for building systems software*. ACM Trans. Comput. Syst. 26(1):1-42, 2008.

(Cuesta & Romay, 2010) C.E. Cuesta, M.d.P. Romay. *Elements of Self-adaptive Systems - A Decentralized Architectural Perspective*. In: Self-Organizing Architectures. Lecture Notes on Computer Science Series, vol. 6090, pp. 1-20. Springer-Verlag, Berlin Heidelberg, July 2010.

(Cuesta et al., 2001) C.E. Cuesta, P.d.l. Fuente, M. Barrio-Solárzano. *Dynamic Coordination Architecture through the use of Reflection*. ACM Symposium on Applied Computing (SAC'01). Las Vegas, Nevada, USA, 2001.

(Cuesta et al., 2002) C.E. Cuesta, P.d.l Fuente, M. Barrio-Solórzano, M.E. Beato. *Introducing Reflection in Architecture Description Languages*. 3rd IEEE/IFIP Conference on Software Architecture (WICSA'02). Montréal, Québec, Canada, August 25-30, 2002.

(Cuesta et al., 2004) C.E. Cuesta, P. Romay, P.d.l. Fuente, M. Barrio-Solórzano. *Reflection-Based, Aspect-Oriented Software Architecture*. In proc. of: First European Workshop on Software Architecture (EWSA'04). LNCS, 3047. Springer, 2004.

(Cuesta et al., 2005) C.E. Cuesta, M.d.P. Romay, P.d.l Fuente, M. Barrio-Solárzano. *Architectural aspects of architectural aspects*. In proc. of: 2nd European Workshop on Software Architecture (EWSA'05). LNCS, vol. 3527. Springer, 2005.

(Cuesta et al., 2006) C.E. Cuesta Quintero, M.d.P. Romay, P.d.l Fuente, M. Barrio-Solórzano. *Temporal Superimposition of Aspects for Dynamic Software Architecture*. In proc. of: 8th IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06). LNCS, vol. 4037, pp. 93-107. Springer, 2006

(Cuesta, 2002) C.E. Cuesta. *Reflection-Based Dynamic Software Architecture*. PhD Thesis, Department of Computing Science, Universidad de Valladolid, 2002 (in Spanish).

(Dashofy et al., 2001) E.M. Dashofy, A. van der Hoek, R.N. Taylor. *A Highly-Extensible, XML-Based Architecture Description Language.* Working IEEE/IFIP Conference on Software Architecture (WICSA'01). August, 2001.

(Dashofy et al., 2002) E.M. Dashofy, A. van der Hoek, R.N. Taylor. *Towards Architecture-Based Self-Healing Systems*. In proc. of: First Workshop on Self-Healing Systems (WOSS'02). Charleston, South Carolina, 2002.

(David & Ledoux, 2005) P. David and T. Ledoux. *WildCAT: a generic framework for context-aware applications*. In: 3rd Int. Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC'05). Grenoble, France, 2005.

(David & Ledoux, 2006) P. David, T. Ledoux. *An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components*. In proc. of: 5th Symposium on Software Composition (SC'06). Vienna, Austria, 2006.

(David et al., 2009) P. David, T. Ledoux, M. Léger, T. Coupaye. *FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures*. Annales des Télécommunications 64(1-2):45-63, 2009.

(De Lucia et al., 2009) A. De Lucia, V. Deufemia, C. Gravino, M. Risi. *Behavioral Pattern Identification through Visual Language Parsing and Code Instrumentation*. In proc of: 13th European Conference on Software Maintenance and Reengineering (CSMR'09). Kaiserslautern, Germany, 2009.

(Demers & Malenfant, 1995) F-N. Demers, J. Malenfant. *Reflection in Logic, Functional and Object-Oriented Programming: a Short Comparative Study*. In: IJCAI'95 workshop on Reflection and Meta-Level Architectures and their Applications in Artifficial Intelligence, pp. 29-38. Montreal, 1995.

(DeRemer & Kron, 1976) F. DeRemer, H. Kron. *Programming in the Large vs Programming in the Small*. IEEE Transactions on Software Engineering, 2(2):80-86, 1976.

(Dijkstra, 1974) E.W. Dijkstra. *A Discipline of Programming*. EWD. 477, Neuen, The Netherlands, 30 August 1974.

(Dodig-Crnkovic, 2002) G. Dodig-Crnkovic. *Scientific Methods in Computer Science*. In: Conf. for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, 2002.

(Douence & Le Botlan, 2005) R. Douence, D. Le Botlan. *Towards a taxonomy of AOP semantics*. Technical report AOSD Europe, milestone M8.1. July, 2005. Available in: http://www.emn.fr/z-info/dlebotla/papers/dl05.pdf

(Dowling & Cahill, 2001) J. Dowling, V. Cahill. *The K-Component Architecture Meta-Model for Self-Adaptive Software*. In proc. of 3rd Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection'2001). LNCS vol. 2192, pp. 81-88. Springer-Verlag, Sep. 2001.

(Dowling & Cahill, 2004) J. Dowling, V. Cahill. *Self-managed decentralised systems using K-components and collaborative reinforcement learning*. In proc. of the 1st ACM SIGSOFT Workshop on Self-Managed Systems. Newport Beach, California, Oct. 31 – Nov. 01, 2004.

(Easterbrook et al., 2008) S. Easterbrook, J. Singer, M.A. Storey, D. Damian. *Selecting Empirical Methods for Software Engineering Research*. Guide to Advanced Empirical Software Engineering. Springer, 2008.

(Ehrig et al., 2006) H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, 2006.

(Elrad et al., 2001) T. Elrad, R.E. Filman, A. Bader. *Aspect-Oriented Programming: An Introduction*. Communication of the ACM, Vol. 44, No. 10, October 2001.

(Endler & Wei, 1992) M. Endler & J. Wei. *Programming Generic Dynamic Reconfigurations for Distributed Applications*. In: First International Workshop on Configurable Distributed Systems. London, UK, 1992.

(Engels et al., 2001) G. Engels, R. Heckel, J.M. Küster. *Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model.* In: Gogolla, M., Kobryn, C. (eds.): «UML» 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools. LNCS, vol. 2185, pp. 272-286. Springer, Heidelberg, 2001.

(Engels et al., 2002) G. Engels, R. Heckel, J.M. Küster, L. Groenewegen. *Consistency-Preserving Model Evolution through Transformations*. In: Jézéquel, J.M., Hußmann, H., Cook, S. (eds.): «UML» 2002 - The Unified Modeling Language. LNCS, vol. 2460, pp. 212-227. Springer, Heidelberg, 2002.

(Fabry, 1976) R.S. Fabry. *How to design a system in which modules can be changed on the fly*. In: 2nd International Conference on Software Engineering (ICSE'76). San Francisco, California, USA, 1976.

(Fiadeiro & Maibaum, 1997) J.L. Fiadeiro, T.S.E. Maibaum. *Categorical Semantics of Parallel Program Design*. Science of Computer Programming, 28:111-138, 1997.

(Filman et al., 2004) R.E. Filman, T. Elrad, S. Clarke, M. Aksit. *Aspect-Oriented Software Development*. Addison Wesley Professional, 2004.

(Garcia et al., 2006) A. Garcia, C. Chavez, T. Batista, C. Sant'anna, U. Kulesza, A. Rashid, C. Lucena. *On the Modular Representation of Architectural Aspects*. In proc. of 3^{rd} European Workshop on Software Architecture (EWSA'06), pp. 82-97. 2006.

(Garlan & Perry, 1995) D. Garlan, D.E. Perry. *Introduction to the Special Issue on Software Architecture*. IEEE Transactions on Software Engineering, 21(4), April 1995.

(Garlan et al., 1997) D. Garlan, R.T. Monroe, D. Wile. *ACME: An Architectural Interchange Language*. Proceedings of the 1997 Conference of the Centre For Advanced Studies on Collaborative Research. Toronto, Ontario, Canada, November 1997.

(Garlan et al., 2004) D. Garlan, S. Cheng, A. Huang, B. Schmerl, P. Steenkiste. *Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure*. Computer, 37:46-54. IEEE, 2004.

(Garlan, 2001) D. Garlan. *Software Architecture*. Wiley Encyclopedia of Software Engineering. John Wiley & Sons, 2001.

(Georgiadis et al., 2002) I. Georgiadis, J. Magee, J. Kramer. *Self-organising software architectures for distributed systems*. In proc. of: First Workshop on Self-Healing Systems (WOSS'02). Charleston, South Carolina, 2002.

(German-Rivera et al., 1996) J. German-Rivera, A.A. Danylyszyn, C.B. Weinstock, L.R. Sha, M.J. Gagliardi. *An Architectural Description of the Simplex Architecture*, Tech. Report CMU/SEI-96-TR-006, 1996.

(Gilb, 1981) T. Gilb. *Evolutionary development*. SIGSOFT Software Engineering Notes, Vol. 6, No. 2, pp. 17. ACM, 1981.

(Glass et al., 2004) R.L. Glass, I. Vessey, V. Ramesh. *Research in software engineering: an analysis of the literature*. Information and Software Technology, Volume 44, Issue 8, pp. 491-506. Elsevier, 2002.

(Gomaa & Hussein, 2004) H. Gomaa, M. Hussein. *Software reconfiguration patterns for dynamic evolution of software architectures*. In: 4th Working International Conference on Software Architecture (WICSA'04). IEEE, 2004.

(Gomez & Ramos, 2010) A. Gómez, I. Ramos. *Cardinality-Based Feature Modeling and Model-Driven Engineering: Fitting them Together*. In proc. of: 4th Int. Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'10), pp. 61-68. Linz, Austria, January 27-29, 2010.

(Gray & Hjálmtýsson, 1998) R. Gray, G. Hjálmtýsson. *Dynamic c++ classes: A lightweight mechanism to update code in a running program*. In USENIX Technical Conference, 1998.

(Gray & Reuter, 1993) J. Gray, A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kauffman, 1993.

(Greenfield et al., 2004) J. Greenfield, K. Short, S. Cook and S. Kent. *Software Factories*. Wiley Publising Inc., 2004.

(Greenwood & Blair, 2006) P. Greenwood and L. Blair. *A Framework for Policy Driven Auto-adaptive Systems Using Dynamic Framed Aspects*. Transactions on Aspect-Oriented Software Development II. LNCS, vol. 4242, pp. 30-65. Springer, 2006.

(Guillén-Martín, 2007) J. Guillén-Martín. *Automatic Code Generation and Execution of Aspect-Oriented Software Architectures*. Ms Science Thesis. Faculty of Computer Science, Universidad Politécnica de Valencia, July 2007 (in Spanish).

(Gupta & Jalote, 1993) D. Gupta, P. Jalote. *On-line software version change using state transfer between processes*. Software-Practice and Experience, 23(9):949–964, 1993.

(Gupta et al., 1996) D. Gupta, P. Jalote, G. Barua. *A Formal Framework for On-line Software Version Change*. IEEE Trans. Softw. Eng., Vol. 22, No. 2, pp. 120-131, Feb. 1996.

(Gustavsson et al., 2004) J. Gustavsson, T. Staijen, U. Aßmann. *Runtime Evolution as an Aspect*. In proc. of 1st Int. Workshop on Foundations of Unanticipated Software Evolution (FUSE'04). Barcelona, Spain, 2004.

(Hammer, 2009) M. Hammer. *How to Touch a Running System: Reconfiguration of Stateful Components*. PhD. Thesis, Faculty of Mathematics, Computer Science and Statistics, LMU München, June 2009.

(Harrison et al., 2002) W.H. Harrison, H.L. Ossher, P.L. Tarr. *Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition*. Technical Report RC22685, Thomas J. Watson Research Center, IBM, 2002.

(Heckel, 2006) R. Heckel. *Graph Transformation in a Nutshell*. School on Foundations of Visual Modelling Techniques (FoVMT 04). ENTCS, vol. 148(1), pp. 187-198. Elsevier, 2006.

(Heimerdinger & Weinstock, 1992) W.L. Heimerdinger, C.B. Weinstock. *A Conceptual framework for System Fault Tolerance*. Technical Report CMU/SEI-92-TR-033. Carnegie Mellon University, 1992.

(Hellerstein et al., 2004) J.L. Hellerstein, Y. Diao, S. Parekh, D.M. Tilbury. *Feedback Control of Computing Systems*. Wiley & Sons, 2004.

(Hervás-Muñoz, 2009) D. Hervás-Muñoz. *Dynamic Evolution of Aspect-Oriented Components in .NET*. Ms Science Thesis. Faculty of Computer Science, Universidad Politécnica de Valencia, February 2009 (in Spanish).

(Hevner et al., 2004) A.R. Hevner, S.T. March, J. Park, S. Ram. *Design science in information systems research*. MIS Quarterly, 28, pp. 75-105, 2004.

(Hicks & Nettles, 2005) M. Hicks, S. Nettles. *Dynamic software updating*. ACM Transactions on Programming Languages and Systems (TOPLAS), 27(6):1049-1096, November 2005.

(Hicks, 2001) M. Hicks. *Dynamic Software Updating*. PhD thesis, Dept. of Computer and Information Science, University of Pennsylvania, June 2001.

(Hirsch et al., 1998) D. Hirsch, P. Inverardi, U. Montanari. *Graph grammars and constraint solving for software architecture styles*. In: 3rd Int. Software Architecture Workshop (ISAW-3). ACM Press, 1998.

(Hofmeister et al., 1999) C. Hofmeister, R.L. Nord, D. Soni. *Applied Software Architecture*. Addision Wesley Longman, 1999.

(Huebscher & McCann, 2008) M.C. Huebscher, J.A. McCann. *A survey of autonomic computing-degrees, models, and applications*. ACM Computer Surveys, 40(3). ACM, 2008.

(IBM, 2001) IBM. *Autonomic Computing: IBM's perspective on the state of information technology*. Available in: http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf

(IBM, 2006) IBM. *An architectural blueprint for autonomic computing*. White paper, 4th edition: http://www.research.ibm.com/autonomic

(IEEE, 2000) IEEE Software Engineering Standards Committee. *Recommended Practices for Architectural Description of Software-Intensive Systems (IEEE Std 1471-2000)*. IEEE Computer Society, Sept. 2000.

(IFR, 2009) International Federation of Robotics, Statistical Department. World Robotics 2009 Study. Available online at: http://www.worldrobotics.org/downloads/2009_executive_summary.pdf

(ISO/IEEE, 2006) International Standard *ISO/IEC 14764 IEEE Std 14764-2006. Software Engineering – Software Life Cycle Processes – Maintenance (Revision of IEEE Std 1219-1998)*. September, 2006.

(Jackson, 1999) D. Jackson. *Alloy: A Lightweight Object Modelling Notation*. MIT Lab for Computer Science, July 1999.

(Jacobson & Ng 2003) I. Jacobson & Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Addison Wesley Professional, 2005.

(Jacobson et al., 1997) I. Jacobson, M. Griss, P. Jonsson. *Software Reuse. Architecture, Process and Organization for Business Success*. Addison-Wesley 1997.

(Johnson, 2010) C. Johnson. *What is research in computing science?* Computer Science Dept., Glasgow University. Electronic resource: http://www.dcs.gla.ac.uk/~johnson/teaching/research_skills/research.html

(Keeney & Cahill, 2003) J. Keeney and V. Cahill. *Chisel: A Policy-Driven, Context-Aware, Dynamic Adaptation Framework*. In: 4th IEEE International Workshop on Policies for Distributed Systems and Networks. June 04-06, 2003.

(Keeney, 2004) J. Keeney. *Completely Unanticipated Dynamic Adaptation of Software*. PhD Thesis. Trinity College, University of Dublin, 2004.

(Kephart & Chess, 2003) J.O. Kephart & D.M. Chess. *The Vision of Autonomic Computing*. Computer, 36(1):41-50. IEEE, 2003.

(Kiczales et al., 1997) G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. Loingtier, J. Irwin. *Aspect-Oriented Programming*. In proc. of: 11th European Conference on Object-Oriented Programming (ECOOP'97), pp. 220-242. Jyväskylä, Finland, June 9-13 1997.

(Kiczales et al., 2001) G. Kiczales, E. Hilsdale, J. Huguin, M. Kersten, J. Palm, W.G. Griswold. *An Overview of AspectJ*. In: 15th European Conference on Object-Oriented Programming (ECOOP'01). Lecture Notes in Computer Science, Vol. 2072. Springer-Verlag, 2001.

(Kirby et al., 1992) G.N.C. Kirby, R.C.H. Connor, Q.I. Cutts, A. Dearle, A.M. Farkas, R. Morrison. *Persistent hyperprograms.* 5th Int. Conf. on Persistent Object Systems. Springer, Berlin, 1992.

(Kon et al., 2002) F. Kon, F. Costa, G. Blair, R.H. Campbell. *The Case for Reflective Middleware*. Communications of the ACM 45(6):33-38. ACM, 2002.

(Kramer & Magee, 1990) J. Kramer & J. Magee. *The Evolving Philosophers Problem: Dynamic Change Management*. IEEE Transactions on Software Engineering, 16(11):1293-1306. IEEE, 1990.

(Kramer & Magee, 2007) J. Kramer and J. Magee. *Self-managed systems: an architectural challenge*. In proc. of: ICSE - Future of Software Engineering (FOSE'07), pp. 259–268. IEEE, 2007.

(Kramer & Magge, 1985) J. Kramer, J. Magee. *Dynamic Configuration for Distributed Systems*. IEEE Transactions on Software Engineering, Vol. 11, No. 4, pp. 424-436, 1985.

(Lehman, 1980) M.M. Lehman. *Programs, life cycles, and laws of software evolution*. Proceedings of the IEEE, Vol. 68, No. 9, pp. 1060-1076, Sept. 1980.

(Lin et al., 2005) P. Lin, A. MacArthur, J. Leaney. *Defining Autonomic Computing: A Software Engineering Perspective*. In proc. of: Australian Conf. Software Engineering (ASWEC'05), pp. 85-97. IEEE CS, 2005.

(Llavador & Canos, 2007) M. Llavador, J.H. Canós. *A Framework for the Generation of Transformation Templates*. Research and Advanced Technology for Digital Libraries, 11th European Conference (ECDL'07). Lecture Notes in Computer Science, vol. 4675, pp. 501-504. Springer 2007.

(Maes, 1987) P. Maes. *Concepts and Experiments in Computational Reflection*. In: SIGPLAN Notices, Vol. 22, No. 12, pp. 147-155. ACM Press, New York, NY, USA, 1987.

(Magee & Kramer, 1996) J. Magee & J. Kramer. *Dynamic Structure in Software Architectures*. ACM Software Engineering Notes, 21(6):3-14. ACM, November 1996.

(Magee et al., 1989) J. Magee, J. Kramer, M.S. Sloman. *Constructing distributed systems in Conic*. Transactions on Software Engineering, 6(15):663–675, June 1989.

(Magee et al., 1995) J. Magee, N. Dulay, S. Eisenbach, J. Kramer. *Specifying Distributed Software Architectures*. In: 5th European Software Engineering Conference (ESEC'95). Sitges, Spain, 1995.

(Malabarba et al., 2000) S. Malabarba, R. Pandey, J. Gragg, E. Barr, J.F. Barnes. *Runtime support for type-safe dynamic Java classes*. In: Bertino, E. (ed.): ECOOP 2000- Object-Oriented Programming. LNCS, vol. 1850, pp. 337-361. Springer, Heidelberg, 2000.

(Malenfant et al., 1996) J. Malenfant, M. Jacques, F-N. Demers. *A Tutorial on Behavioural Reflection and its Implementation*. In: Reflection'96, pp. 1-20, 1996.

(Mamei & Zambonelli, 2009) M. Mamei, F. Zambonelli. *Programming pervasive and mobile computing applications: The TOTA approach*. ACM Trans. Software Engineering Methodology 18(4):1-56. July, 2009.

(March & Smith, 1995) S.T. March, G. Smith. *Design and Natural Science Research on Information Technology*. Decision Support Systems 15(4), pp. 251-266. December, 1995.

(Martin, 2002) R.C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

(McKinley et al., 2004) P.K. McKinley, S.M. Sadjadi, E.P. Kasten, and B.H.C. Cheng. *Composing Adaptive Software*. Computer, 37(7):56-64. IEEE, July 2004.

(McKinley et al., 2004a) P.K. McKinley, S.M. Sadjadi, E.P. Kasten, B.H.C. Cheng. *A Taxonomy of Compositional Adaptation*. Technical Report MSU-CSE-04-17, Dept. Computer Science and Engineering, Michigan State Univ., 2004.

(McKinley et al., 2008) P. McKinley, B.H.C. Cheng, C. Ofria, D. Knoester, B. Beckmann, H. Goldsby. *Harnessing Digital Evolution*. IEEE Computer, 41(1):54-63. Jan, 2008.

(Medvidovic & Taylor, 2000) N. Medvidovic, R.N. Taylor. *A Classification and Comparison Framework for Software Architecture Description Languages*. Transactions on Software Engineering 26(1):70-93. IEEE, 2000.

(Mens & Wermelinger, 2002) T. Mens and M. Wermelinger. *Separation of concerns for software evolution*. Journal of Software Maintenance and Evolution, 14(5):311-315. Wiley, 2002.

(Mens, 2008) T. Mens. *Introduction and Roadmap: History and Challenges of Software Evolution*. T. Mens, S. Demeyer (eds.): Software Evolution, chapter 1. Springer, 2008.

(Mickan et al., 2004) K. Mickan, R. Morrison, G. Kirby, D. Balasubramaniam, E. Zirintsis. *Using generative programming to visualise hypercode in complex and dynamic systems*. 27th Australasian Conference on Computer Science. Australian Computer Society, Darlinghurst, Australia, 377-386.

(Mikunov, 2003) A. Mikunov. *Rewrite MSIL Code on the Fly with the .NET Framework Profiling API*. MSDN Magazine, September 2003.

(Millán-Belda, 2006) C. Millán-Belda. *Development of Distributed and Mobile Applications from an Aspect-Oriented Architectural Model*. Master Science Thesis. Faculty of Computer Science, Universidad Politécnica de Valencia, September 2006 (in Spanish).

(Milner, 1993) R. Milner. *The Polyadic π-Calculus: A Tutorial*. Laboratory for Foundations of Computer Science Deptartment. University of Edinburgh, 1993.

(Mittermeir, 2001) R.T. Mittermeir. *Software evolution: Let's sharpen the terminology before sharpening (out-of-scope) tools*. In: 4th International Workshop on Principles of Software Evolution (IWPSE'01). ACM, New York, NY, 2001.

(Moazami-Goudarzi, 1999) K. Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems*. PhD thesis, Imperial College, London, March 1999.

(Monroe, 98) R.T. Monroe. *Capturing Software Architecture Design Expertise With Armani*. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science, 1998.

(Morrison et al., 2004) R. Morrison, G.N.C. Kirby, D. Balasubramaniam, K. Mickan, F. Oquendo, S. Cîmpan, B.C. Warboys, B. Snowdon, R.M. Greenwood. *Support for evolving software architectures in the ArchWare ADL*. 4th Working IEEE/IFIP Conf. on Software Architecture (WICSA'04). Oslo, Norway, 2004.

(Morrison et al., 2007) R. Morrison, D. Balasubramaniam, G. Kirby, K. Mickan, B. Warboys, R.M. Greenwood, I. Robertson, R. Snowdon. *A Framework for Supporting Dynamic Systems Co-Evolution*. Automated Software Engineering, 14(3):261-292. Springer, 2007.

(Muller et al., 2008) H.A. Müller, H.M. Kienle, U. Stege. *Autonomic Computing Now You See It, Now You Don't*. In: ISSSE 2008. LNCS vol. 5413, pp. 32-54. Springer, 2008.

(Navarro, 2008) E. Navarro. *ATRIUM – Architecture Traced from RequIrements by applying a Unified Methodology*. PhD Thesis, University of Castilla-La Mancha, May 2007.

(Navasa et al., 2007) A. Navasa, M.A. Pérez-Toledano, J.M. Murillo. *AspectLEDA: Extending an ADL with Aspectual Concepts*. In proc. of: 1st European Conference on Software Architecture (ECSA'07). LNCS, vol. 4758, pp. 330–334. Springer, 2007.

(Navasa et al., 2009) A. Navasa, M.A. Perez-Toledano, J.M. Murillo. *An ADL dealing with aspects at software architecture stage*. Information and Software Technology, 51(2):306-324. Elsevier, February 2009.

(Nicoara et al., 2008) A. Nicoara, G. Alonso, T. Roscoe. *Controlled, Systematic, and Efficient Code Replacement for Running Java Programs*. In: ACM SIGOPS Operating Systems Review, Vol. 42, No. 4. May 2008.

(OMG, 2002) Object Management Group. *Meta-Object Facility 1.4 Specification*. Technical Report formal/2002-04-03, 2002. In: http://www.omg.org/technology/documents/formal/mof.htm

(OMG, 2003) Object Management Group. *Model Driven Architecture Guide*, 2003. In: http://www.omg.org/docs/omg/03-06-01.pdf

(Ommering et al., 2000) R. van Ommering, F. van der Linden, J. Kramer, J. Magee. *The Koala Component Model for Consumer Electronics Software*. IEEE Computer, 33(3):78-85, Mar 2000.

(Oquendo et al., 2004) F. Oquendo, B. Warboys, R. Morrison, R. Dindeleux, F. Gallo, H. Garavel, C. Occhipinti. *ArchWARE: Architecting Evolvable Software*. In proc. of: European Workshop in Software Achitecture (EWSA 2004). Lecture Notes in Computer Science, vol. 3047, pp. 257-271. Springer, 2004.

(Oreizy et al., 1998) P. Oreizy, N. Medvidovic, R.N. Taylor. *Architecture-based runtime software evolution*. 20th international Conference on Software Engineering (ICSE'98). IEEE, 1998.

(Oreizy et al., 1999) P. Oreizy, M. Gorlick, R.N. Taylor, D. Heimbinger, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum and A.L. Wolf. *An*

Architecture-Based Approach to Self-Adaptive Software. Intelligent Systems, 14:54-62. IEEE, 1999.

(Parnas, 1972) D.L. Parnas. *On the Criteria To Be Used in Decomposing Systems Into Modules*. Communications of the ACM, Vol. 15, No. 12, pp. 1053-1058. December, 1972.

(Parnas, 1994) D.L. Parnas. *Software aging*. In proc. of: International Conference on Software Engineering (ICSE'94). IEEE, 1994.

(Pedersen et al., 2006) S.M. Pedersen, S. Fountas, H. Have, B.S. Blackmore. *Agricultural robots–system analysis and economic feasibility*. Precision agriculture 7(4):295-308. Springer, 2006.

(Pérez & Cuesta, 2007) J. Pérez, C.E. Cuesta. *Aspect-oriented connectors for coordination*. In: International Workshop on Synthesis and Analysis of Component Connectors (SYANCO'07), in Conjunction with the 6th ESEC/FSE Joint Meeting. Dubrovnik, Croatia, September 3-4, 2007.

(Pérez et al., 2005a) J. Pérez, N. Ali, C. Costa-Soria, J.A. Carsí, I. Ramos. *Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*. In proc. of: 3rd International Conference on .NET Technologies, pp. 97-108. Pilsen, Czech Republic, June 2005.

(Pérez et al., 2005b) J. Pérez, N. Ali, J.A. Carsí, I. Ramos. *Dynamic Evolution in Aspect-Oriented Architectural Models*. In proc. of: 2nd European Workshop on Software Architecture (EWSA'05). LNCS, Vol. 3527, pp. 59-76. Springer, Pisa, Italy, 2005.

(Pérez et al., 2006) J. Pérez, N. Ali, J.A. Carsí, I. Ramos. *Designing Software Architectures with an Aspect-Oriented Architecture Description Language*. In proc. of: 9th Int. Symp. on Component-Based Software Engineering (CBSE'06). LNCS, Vol. 4063, pp. 123-138. Springer, 2006.

(Pérez et al., 2007) J. Pérez, C. Costa, J.A. Carsí, I. Ramos. *Verification of Aspect-Oriented Architectural Models*. In proc. of: XII Conference on Software Engineering and Databases (JISBD'07), pp. 167-176. Zaragoza, 11-14 sep 2007. ISBN 978-84-9732-595-0.

(Pérez et al., 2007a) J. Pérez, C. Costa, J.A. Carsí, I. Ramos. *PRISMA CASE*. In proc. of: XII Conference on Software Engineering and Databases (JISBD'07), pp. 399-400. Zaragoza, 11-14 sep 2007.

(Pérez et al., 2008) J. Pérez, I. Ramos, J.A. Carsí. *Taking Advantage of COTS for Developing Aspect-Oriented Software Architectures*. In: 15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems (ECBS'08), pp. 245-254. Belfast, Northern Ireland, 31 March - 4 April, 2008.

(Pérez et al., 2008) J. Pérez, N. Ali, J.A. Carsí, I. Ramos, B. Álvarez, P. Sánchez, J.A. Pastor. *Integrating Aspects in Software Architectures: PRISMA Applied to Robotic Tele-operated Systems*. Journal of Information & Software Technology, 50(9-10):969-990. Elsevier, 2008.

(Pérez, 2006) J. Pérez. *PRISMA: Aspect-Oriented Software Architectures*. PhD Thesis, Universidad Politécnica de Valencia, December 2006.

(Perez-Toledano et al., 2007) M.A. Perez-Toledano, A. Navasa, J.M. Murillo, C. Canal. *TITAN: a Framework for Aspect Oriented System Evolution*. In: International Conference on Software Engineering Advances (ICSEA'07). IEEE, 2007.

(Perry & Wolf, 1992) D.E. Perry & A.L. Wolf. *Foundations for the Study of Software Architecture*. Software Engineering Notes, 17(4):40-52. ACM, 1992.

(Perry, 2008) D.E. Perry. *Issues in Architecture Evolution: Using Design Intent in Maintenance and Controlling Dynamic Evolution*. In proc. of: European Conference on Software Architecture (ECSA'08). Lecture Notes on Computer Science, vol. 5292. Springer, 2008.

(Pessemier et al., 2008) N. Pessemier, L. Seinturier, L. Duchien, T. Coupaye. *A Component-Based and Aspect-Oriented Model for Software Evolution*. International Journal of Computer Applications in Technology (IJCAT), 31(1-2):94-105. Inderscience, 2008.

(Phung-Khac et al., 2008) A. Phung-Khac, M.T. Segarra, J.M. Gilliot, A. Beugnard. *Dynamic Composition and Adaptation in Adapt-Medium*. In proc. of: Workshop on Autonomic and SELF-Adaptive Systems (WASELF'08), pp. 43-52. San Sebastián, Spain, September 8th, 2008.

(Phung-Khac et al., 2010) A. Phung-Khac, J. Gilliot, Maria-Teresa Segarra, A. Beugnard, E. Kaboré. *Modelling Changes and Data Transfers for Architecture-Based Runtime Evolution of Distributed Applications*. In proc. of: 4th European Conference on Software Architecture (ECSA'2010). LNCS, vol. 6285, pp. 392-400. Springer, 2010.

(Pinto et al., 2005) M. Pinto, L. Fuentes and J.M. Troya. *A Dynamic Component and Aspect-Oriented Platform*. In: The Computer Journal, Vol. 48, No. 4, pp. 401-420. Oxford University Press, 2005.

(Pissias & Coulson, 2008) P. Pissias, G. Coulson. *Framework for quiescence management in support of reconfigurable multi-threaded component-based systems*. IET Software 2(4): 348-361, 2008.

(Plasil et al., 1998) F. Plasil, D, Balek, R. Janecek. *SOFA/DCUP: Architecture for component trading and dynamic updating*. International Conference on Configurable Distributed Systems (ICCDS '98), pp 43-52. IEEE, 1998.

(Pollack, 2006) J.B. Pollack. *Mindless Intelligence*. IEEE Intelligent Systems Vol. 21, No. 3, pp. 50-56, 2006.

(Popovici et al., 2002) A. Popovici, T. Gross, G. Alonso. *Dynamic Weaving for Aspect-Oriented Programming*. In: First Intern. Conf. on Aspect-Oriented Software Development (AOSD'02). Enschede, The Netherlands, 2002.

(Popovici et al., 2003) A. Popovici, G. Alonso, T. Gross. *Just-In-Time Aspects: Efficient DynamicWeaving for Java*. In proc. of 2nd International Conference on Aspect-Oriented Software Development (AOSD'03), Boston, USA, March 2003.

(Pressman, 2005) R.S. Pressman. *Software Engineering: A Practitioners Approach* (6th Edition). McGraw-Hill, 2005.

(Raheja et al., 2010) R. Raheja, S. Wen-Cheng, D. Garlan and B. Schmerl. *Improving Architecture-Based Self-Adaptation Using Preemption*. In: Self-Organizing Architectures. Lecture Notes on Computer Science Series, vol. 6090, pp. 21-37. Springer-Verlag, Berlin Heidelberg, July 2010.

(Rajan & Sullivan, 2003) H. Rajan, K. Sullivan. *EOS: Instance-Level Aspects for Integrated System Design*. In: 9th European Software Engineering Conference held jointly with 11th Intern. Symp. on Foundations of Software Engineering (ESEC-FSE'03). Helsinki, Finland, Sept. 2003.

(Rasche & Polze, 2003) A. Rasche, A. Polze. *Configuration and Dynamic Reconfiguration of Component-Based Applications with Microsoft .NET*. In proc. of: 6th IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03). Hakodate, Japan, 2003.

(Ritzau & Andersson, 2000) T. Ritzau and J. Andersson. *Dynamic Deployment of Java Applications*. In proc. of: Java for Embedded Systems Workshop. London, 2000.

(Rogers et al., 2009) A. Rogers, N.R. Jennings, A. Farinelli. *Self-Organising Sensors for Wide Area Surveillance using the Max-Sum Algorithm*. In proc. of: WICSA/ECSA Workshop on Self-Organizing Architectures (SOAR'09). Cambridge, UK, 2009.

(Rombach, 2009) H.D. Rombach. *Design for Maintenance - Use of Engineering Principles and Product Line Technology*. In: 13th European Conf. on Software Maintenance and Reengineering (CSMR'09). Kaiserslautern, Germany, 2009.

(Rutle, 2010) A. Rutle. *Diagram Predicate Framework*. PhD Thesis, University of Bergen, Norway, September 2010.

(Salehie & Tahvildari, 2009) M. Salehie, L. Tahvildari. *Self-adaptive software: Landscape and research challenges*. ACM Transactions on Autonomous and Adaptive Systems 4(2):1-42, May. 2009.

(Sanchez et al., 1998) F. Sanchez, J. Hernandez, J.M. Murillo, E. Pedraza. *Runtime adaptability of synchronization constraints in COOLs*. In: 2nd ECOOP Workshop of Aspect Oriented Programming, 1998.

(Santiago-Perez et al., 2009) J. Santiago-Pérez, C.E. Cuesta, S. Ossoswki. *The Agreement as an Adaptive Architecture for Open Multi-Agent Systems*. 2nd Workshop on Autonomic and SELF-adaptive Systems (WASELF'09). San Sebastián, Spain, 8th September 2009.

(Santos et al., 2001) J. Santos, R. J. Duro, J. A. Becerra, J. L. Crespo and F. Bellas. *Considerations in the application of evolution to the generation of robot controllers*. In: Information Sciences, vol. 133(3-4): 127-148. April 2001.

(Schlegel et al., 2009) C. Schlegel, T. Hassler, A. Lotz, and A. Steck. *Robotic software systems: from code-driven to model-driven designs*. In Proc. of: International Conference on Advanced Robotics (ICAR 2009). IEEE, 2009.

(Schult & Polze, 2003) W. Schult, A. Polze. *Speed vs. Memory Usage-an Approach to Deal with Contrary Aspects*. 2nd Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), International Conference on Aspect-Oriented Software Development (AOSD). Boston, Massachusetts, 2003.

(Segal & Frieder, 1989) M.E. Segal, O. Frieder. *Dynamic program updating: a software maintenance technique for minimizing software downtime*. Journal of Software Maintenance, Vol. 1, No. 1, pp. 59-79, 1989.

(Segal & Frieder, 1993) Mark E. Segal, Ophir Frieder. *On-the-Fly Program Modification: Systems for Dynamic Updating*. Software, 10(2):53-65. IEEE, 1993.

(SEI, 2006) Software Engineering Institute. *Ultra-Large-Scale Systems: Software Challenge of the Future*. Technical Report. Carnegie Mellon Univ., Pittsburgh, USA, 2006.

(Selic, 2003) B. Selic. *The pragmatics of model-driven development*. Software, 20(5). IEEE, 2003.

(Serugendo et al., 2006) G.D.M. Serugendo, M.P. Gleizes, A. Karageorgos. *Self-organisation and emergence in MAS: An Overview*. Informatica (Slovenia), 30(1):45-54, 2006.

(Sha et al., 1996) L. Sha, R. Rajkumar, M. Gagliardi. *Evolving dependable real-time systems*. IEEE Aerospace Applications Conference on Reliability and Quality of Design. February 1996.

(Shaw & Clements, 2006) M. Shaw, P. Clements. *The Golden Age of Software Architecture*. IEEE Software 23(2):31-39, March 2006.

(Shaw & Garlan, 1996) M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, NJ, USA, 1996.

(Shaw, 1994) M. Shaw. *Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status.* In proc. of: Workshop on Studies of Software Design. January, 1994.

(Shaw, 2001) M. Shaw. *The Coming-of-age of Software Architecture Research*. In: 23rd International Conference on Software Engineering (ICSE'01). IEEE Computer Society, 2001.

(Simon, 1996) H.A. Simon. *The Sciences of the Artificial* (3rd edition). MIT Press, 1996.

(Smith, 1982) B.C. Smith. *Reflection and Semantics in a Procedural Language*. Technical Report MIT-LCS-TR-272, Massachusetts Institute of Technology, 1982.

(Staudt, 2000) B. Staudt Lerner. *A model for compound type changes encountered in schema evolution*. ACM Transactions on Database Systems, 25(1):83–127, March 2000.

(Stirling, 1992) C. Stirling. *Modal and Temporal Logics*. Handbook of Logic in Computer Science, vol. II. Clarendon Press, Oxford, 1992.

(Suvée et al., 2003) D. Suvée, W. Vanderperren, V. Jonckers. *JAsCo: an Aspect-Oriented Approach Tailored for Component-Based Software Development*. 2nd International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, pp. 21-29, Boston, Massachusetts, March, 2003.

(Sykes et al., 2008) D. Sykes, W. Heaven, J. Magee, J. Kramer. *From goals to components: a combined approach to self-management*. Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08). Leipzig, Germany, 2008.

(Szyperski, 2002) C. Szyperski. *Component Software: Beyond Object Oriented Programming*. ACM Press and Addison Wesley, New York, USA, 2002.

(Tarr et al., 1999) P. Tarr, H. Ossher, W. Harrison, S.M. Sutton Jr. *N Degrees of Separation: Multidimensional Separation of Concerns*. In proc of: 21st Intern. Conf. on Software Engineering (ICSE'99). Los Angeles, CA, USA 16-22 May 1999.

(Taylor & Hoek, 2007) R.N. Taylor, A. van der Hoek. *Software Design and Architecture - The once and future focus of software engineering*. In: ICSE - Future of Software Engineering (FOSE'07), pp. 226-243. IEEE, 2007.

(Taylor et al., 2009) R.N. Taylor, N. Medvidovic, E.M. Dashofy. *Software Architecture: Foundations, Theory and Practice*. Wiley, 2009.

(Tewari & Milenkovic, 2006) V. Tewari, M. Milenkovic. *Standards for Autonomic Computing*. Intel Technology Journal 10(4):275-284. Intel Corporation, 2006.

(Thao et al., 2008) C. Thao, E.V. Munson, T.N. Nguyen. *Software Configuration Management for Product Derivation in Software Product Families*. In proc. of: 15th Int. Conf. on Engineering of Computer Based Systems (ECBS'08). Belfast, Northern Ireland, 2008.

(Vanderperren et al., 2005) W. Vanderperren, D. Suvée, M.A. Cibrán, B. De Fraine. *Stateful Aspects in JAsCo*. In proc. of Software Composition (SC 2005), LNCS, vol. 3628, pp. 167-181. Springer, 2005.

(Vandewoude & Berbers, 2005) Y. Vandewoude, Y. Berbers. *Component state mapping for runtime evolution*. In: International Conference on Programming Languages and Compilers. Las Vegas, Nevada, USA, 2005.

(Vandewoude & Berbers, 2005a) Y. Vandewoude, Y. Berbers. *Fresco: Flexible and reliable evolution system for components*. Electronic Notes in Theoretical Computer Science, 127(3):197–205, April 2005.

(Vandewoude et al., 2003) Y. Vandewoude, P. Rigole, D. Urting, Y. Berbers. *Draco: An Adaptive Runtime Environment for Components*. Technical Report CW372, Dept. of Computer Science KULeuven, April 2003.

(Vandewoude et al., 2007) Y. Vandewoude, P. Ebraert, Y. Berbers, T. D'Hondt. *Tranquillity: A low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates*. IEEE Transactions on Software Engineering, 33(12):856-868, 2007.

(Vandewoude, 2007) Y. Vandewoude. *Dynamically updating component-oriented systems*. PhD Thesis, Katholieke Universiteit Leuven, Belgium, 2007.

(Vasseur, 2004) A. Vasseur. *Java Dynamic AOP and Runtime Weaving - How Does AspectWerkz Address It?* In proc. of Dynamic Aspects Workshop, in conjunction with AOSD'04. Lancaster, UK, 2004.

(Wang et al., 2006) Q. Wang, J. Shen, X. Wang, H. Mei. *A Component-Based Approach to Online Software Evolution*. Journal of Software Maintenance and Evolution, 18(3). Wiley 2006.

(Wegner, 1976) P. Wegner. *Research Paradigms in Computer Science*. 2nd Internat. Conf. on Software Engineering (ICSE'76). IEEE, 1976.

(Weise et al., 2009) T. Weise, M. Zapf, M. Ullah Khan, K. Geihs. *Combining Genetic Programming and Model-Driven Development*. In International Journal of Computational Intelligence and Applications (IJCIA), Vol. 8, No. 1, pp. 37-52, 2009.

(Wermelinger & Fiadeiro, 2002) M. Wermelinger, J.L. Fiadeiro. *A Graph Transformation Approach to Software Architecture Reconfiguration*. Sci. Comput. Program., 44(2):133–155, 2002.

(Wermelinger et al., 2001) M. Wermelinger, A. Lopes, J.L. Fiadeiro. *A graph based architectural (re)configuration language*. SIGSOFT Software Engineering Notes 26(5), pp. 21-32. ACM, 2001.

(Wong & Mun, 2005) M.L. Wong, Tuen Mun. *Evolving Recursive Programs by Using Adaptive Grammar Based Genetic Programming*. Genetic Programming and Evolvable Machines, Vol. 6, No. 4, pp. 421-455. Dec 2005.

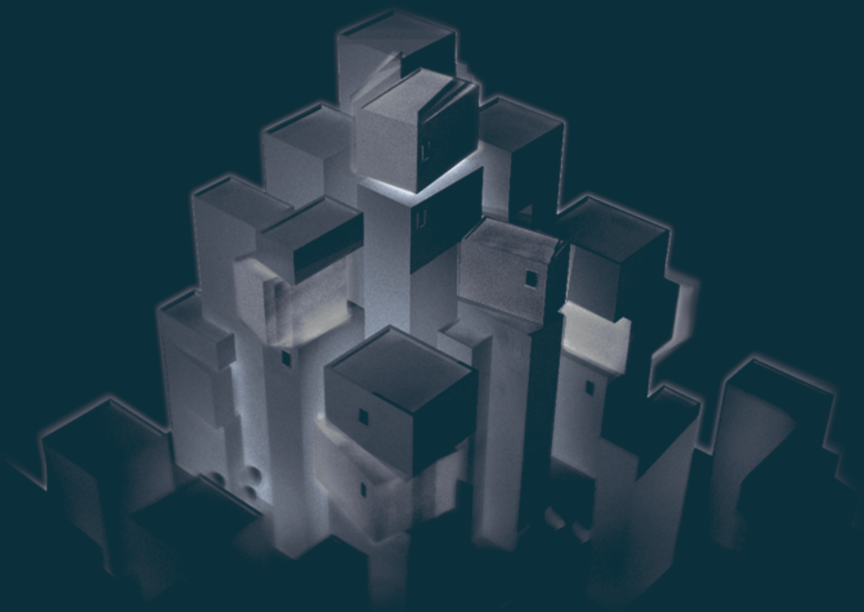
(Xia, 1997) F. Xia. *Software Engineering Research: A Methodological Analysis*. In: Fourth Asia-Pacific Software Engineering and international Computer Science Conference (APSEC'97). IEEE, 1997.

(Yin, 2002) R.K. Yin. *Applications of Case Study Research*. Sage Publications, 3rd edition, 2002.

(Yurcik & Doss, 2001) W. Yurcik, D. Doss. *Achieving Fault-Tolerant Software with Rejuvenation and Reconfiguration*. IEEE Software, 18(4):48-52. IEEE, 2001.

(Zachariadis et al., 2006) S. Zachariadis, C. Mascolo, W. Emmerich. *The SATIN Component System-A Metamodel for Engineering Adaptable Mobile Systems*. IEEE Trans. Software Eng. 32(11): 910-927, 2006.

(Zouari et al., 2010) M. Zouari, M.T. Segarra, F. André. *A Framework for Distributed Management of Dynamic Self-adaptation in Heterogeneous Environments*. In proc. of: 10th IEEE Int. Conf. on Computer and Information Technology (CIT'2010), pp. 265-272. Bradford, UK, June 29-July 1, 2010.



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA